

Développement d'applications avec

Objective Caml

Edition originale
CD-ROM inclus



O'REILLY*

*Emmanuel Chailoux,
Pascal Manoury et Bruno Pagano*

Table of Contents

Prologue

Avant-propos

Préambule

Introduction

Objectifs

Description du langage

Plan de l'ouvrage

Comment obtenir Objective CAML

Logiciels libres

Description du Cédérom

Téléchargement

Installation

Test de l'installation

Noyau du langage

Présentation de la partie I

Programmation fonctionnelle

Introduction

Plan du chapitre

Noyau fonctionnel d'Objective CAML

Déclaration de types et filtrage de motif

Typage, domaine de définition et exceptions

Polymorphisme et valeurs de retour des fonctions

Calculatrice de bureau

Exercices

Résumé

Pour en savoir plus

Programmation impérative

- Introduction
- Plan du chapitre
- Structures de données physiquement modifiables
- Entrées-sorties
- Structures de contrôle
- Ordre d'évaluation des arguments
- Calculatrice à mémoire
- Exercices
- Résumé
- Pour en savoir plus

Styles fonctionnel et impératif

- Introduction
- Plan du chapitre
- Comparaison entre fonctionnel et impératif
- Quel style choisir ?
- Mélange des styles
- Flots de données
- Exercices
- Résumé
- Pour en savoir plus

Interface graphique

- Introduction
- Plan du chapitre
- Mise en oeuvre du module
- Notions de base
- Affichage graphique
- Animation
- Événements
- Calculatrice graphique
- Exercices
- Résumé
- Pour en savoir plus

Applications

Introduction
Requêtes sur une base de données
Interprète BASIC
Démineur

Outils de Développement

Présentation de la partie II
Modes de compilation et portabilité

Introduction
Plan du chapitre
Étapes de compilation
Modes de compilation
Exécutables autonomes
Portabilité et efficacité
Exercices
Résumé
Pour en savoir plus

Bibliothèques

Introduction
Plan du chapitre
Découpage et utilisation des bibliothèques
Bibliothèque préchargée
Bibliothèque standard
Autres bibliothèques de la distribution
Exercices
Résumé
Pour en savoir plus

Récupérateur automatique de mémoire

Introduction
Plan du chapitre
Mémoire et programme
Allocation et récupération mémoire
Récupération automatique
Gestion mémoire en Objective CAML
Module

Module
Exercices
Résumé
Pour en savoir plus

Outils d'analyse des programmes

Introduction
Plan du chapitre
Calcul des dépendances
Outils de mise au point
Profiling
Exercices
Résumé
Pour en savoir plus

Outils d'analyses lexicale et syntaxique

Introduction
Plan du chapitre
Lexique
Syntaxe
Basic revisité
Exercices
Résumé
Pour en savoir plus

Interopérabilité avec C

Introduction
Plan du chapitre
Communication entre C et Objective CAML
Exploration des valeurs Objective CAML en C
Créer et modifier des valeurs Objective CAML en C
Gestion des exceptions en C et en Objective CAML
Programme principal en C
Exercices
Résumé
Pour en savoir plus

Applications

Introduction

Construction d'une interface graphique

Recherche de chemins de moindre coût

Organisation d'Applications

Présentation de la partie III

Programmation modulaire

Introduction

Plan du chapitre

Modules simples

Langage des modules simples

Modules paramétrés

Exemple : gestion de comptes bancaires

Exercices

Résumé

Pour en savoir plus

Programmation par objets

Introduction

Plan du chapitre

Classes, objets et méthodes

Relations entre classes

Autres traits objet

Types et généricité

Sous-typage et polymorphisme d'inclusion

Style fonctionnel

Autres éléments de l'extension objet

Exercices

Résumé

Pour en savoir plus

Comparaison des modèles d'organisation

Introduction

Plan du chapitre

Comparaison modules et objets

Étendre les composants
Organisations mixtes
Exercices
Résumé
Pour en savoir plus

Applications

Introduction
Jeux à deux joueurs
Des robots en goguette

Concurrence et répartition

Présentation de la partie IV
Communication et processus

Introduction
Plan du chapitre
Module
Descripteurs de fichier
Processus
Communication entre processus
Exercices
Résumé
Pour en savoir plus

Programmation concurrente

Introduction
Plan du chapitre
Processus concurrents
Synchronisation des processus
Communication synchrone
Exemple : bureau de Poste
Exercices
Résumé
Pour en savoir plus

Programmation répartie

- Introduction
- Plan du chapitre
- Réseau Internet
- Prises de communication
- Client-serveur
- Protocoles de communication
- Exercices
- Résumé
- Pour en savoir plus

Applications

- Introduction
- Boîte à outils client-serveur
- Les robots de l'aube
- Servlets HTTP

Développement d'applications en Objective CAML

- Introduction
- Éléments d'évaluation
- Autres outils de développement
- Applications réalisées en Objective CAML
- Langages fonctionnels proches
- Langages à Objet : comparaison avec Java
- Avenir des développements en Objective CAML

Conclusion

Annexes

Annexe A Types cycliques

- Introduction
- Types cycliques
- Option

Annexe B Objective CAML 2.99

- Introduction
- Extensions du langage
- Bibliothèque

[Navigateur OCamlBrowser](#)

[Références](#)

[Bibliographie](#)

[Index](#)

[Index des concepts](#)

[Index des éléments du langage](#)

[Parutions O'Reilly](#)

[Contacts](#)

Prologue

Avant-propos

Préambule

La volonté d'écrire un livre sur Objective CAML est venue de l'expérience pédagogique des auteurs dans l'apprentissage des concepts de la programmation à travers le langage Objective CAML. Les étudiants des différents cycles et les ingénieurs en formation permanente de l'Université Pierre et Marie Curie ont par leur dynamisme et leurs critiques grandement fait évoluer notre présentation du langage Objective CAML. Plusieurs exemples de ce livre s'inspirent directement de leurs réalisations.

L'implantation du langage Caml se poursuit depuis quinze ans déjà. Son développement provient des projets Formel puis Cristal de l'INRIA en collaboration avec l'Université Denis Diderot et l'École Normale Supérieure. Les efforts continus, tant théoriques que d'implantation, des chercheurs de ces équipes ont produit au fil des ans un langage de grande qualité. Ils ont su tenir compte de l'évolution constante du domaine en intégrant dans un cadre formel les nouveaux modèles de programmation. Nous espérons contribuer par cet ouvrage à la diffusion que mérite ce travail.

La forme et le fond de ce livre ne seraient pas ce qu'ils sont sans l'aide de nombreux collègues. Ils n'ont pas été rebutés par la relecture de nos premiers manuscrits. Leurs remarques et leur commentaires ont permis d'améliorer cet ouvrage tout au long de sa réalisation. Nous tenons à particulièrement remercier María-Virginia Aponte, Sylvain Baro, Christian Codognet, Hélène Cottier, Guy Cousineau, Pierre Crégut, Titou Durand, Christophe Gonzales, Michelle Morcrette, Christian Queinnec, Attila Raksany et Didier Rémy.

La version HTML de ce livre n'aurait pas vu le jour sans les outils [hevea](#) et [VideoC](#). Un grand merci à leurs auteurs respectifs, Luc Maranget et Christian Queinnec, qui ont toujours répondu dans les plus brefs délais à nos questions et à nos demandes d'évolution.

Introduction

Objectifs

Objective CAML est un langage de programmation : un de plus dira-t-on ! Ils sont en effet déjà nombreux et pourtant il en apparaît constamment de nouveaux. Au delà de leurs disparités, la conception et la genèse de chacun d'eux procèdent d'une motivation partagée : la volonté d'abstraire.

S'abstraire de la machine

En premier lieu, un langage de programmation permet de négliger l'aspect « mécanique » de l'ordinateur; il autorise même à oublier le modèle du microprocesseur ou le système d'exploitation sur lequel sera exécuté le programme.

S'abstraire du modèle opératoire

La notion de fonction que possède sous une forme ou une autre la plupart des langages est empruntée aux mathématiques et non à l'électronique. D'une manière générale, les langages substituent des modèles formels aux conceptions purement calculatoires. Ils y gagnent en expressivité.

Abstraire les erreurs

Il s'agit ici de la tentative de garantir la sûreté d'exécution; un programme ne doit pas se terminer brutalement ou devenir incohérent en cas d'erreur. Un des moyens pour y parvenir est le typage statique fort des programmes et la mise en oeuvre d'un mécanisme d'exceptions.

Abstraire les composants (i)

Les langages de programmation donnent la possibilité de découper une application en différents composants logiciels, plus ou moins indépendants et autonomes. La modularité permet une structuration de plus haut niveau de

l'ensemble d'une application complexe.

Abstraire les composants (ii)

L'existence d'unités de programmes a ouvert la possibilité de leur réutilisation dans d'autres contextes que ceux de leur développement. Les langages à *objets* constituent une autre approche de la réutilisabilité permettant la réalisation très rapide de prototypes.

Objective CAML est un langage récent qui se place dans l'histoire des langages de programmation comme un lointain descendant de LISP ayant su tirer les enseignements de ses cousins en incorporant les principales caractéristiques des autres langages. Il est développé à l'INRIA¹ et s'appuie sur une longue expérience de conception de langages de la famille ML. Objective CAML est généraliste pour l'expression d'algorithmes symboliques ou numériques. Il est orienté objet et possède un système de modules paramétrés. Il permet le développement d'applications concurrentes ou distribuées. Il possède une excellente sûreté à l'exécution grâce à son typage statique, son mécanisme d'exceptions et son récupérateur automatique de mémoire. Il est performant tout en étant portable. Enfin, il dispose d'un riche environnement de développement.

Objective CAML n'a jamais fait l'objet d'une présentation au < < grand public > >. C'est la tâche à laquelle se sont attelés les auteurs en donnant à cet ouvrage trois objectifs :

1. Décrire de manière approfondie le langage Objective CAML, ses bibliothèques et son environnement de développement.
2. Montrer et expliquer quels sont les concepts qui se cachent derrière les styles de programmation utilisables avec Objective CAML.
3. Illustrer par de nombreux exemples comment Objective CAML peut servir de langage de développement pour différentes familles d'applications.

Le but des auteurs est de fournir des éléments de choix entre les différents styles de programmation et les différentes possibilités d'organisation d'un programme pour, en adéquation avec un problème donné, obtenir une maintenance facilitée et une bonne réutilisabilité des composants logiciels développés.

Description du langage

Objective CAML est un langage fonctionnel :

il manipule les fonctions comme étant des valeurs du langage. Celles-ci peuvent être utilisées en tant que paramètres d'autres fonctions ou être retournées comme résultat d'un appel de fonction.

Objective CAML est typé statiquement :

la vérification de la compatibilité entre les types des paramètres formels et des paramètres d'appel est effectuée au moment de la compilation du programme. Dès lors, il n'est pas nécessaire de faire ces vérifications durant l'exécution du programme ce qui accroît son efficacité. En outre, la vérification de type permet d'éliminer la plupart des erreurs introduites par maladresse ou étourderie et contribue à la sûreté de l'exécution.

Objective CAML est polymorphe paramétrique :

une fonction qui n'explore pas la totalité de la structure d'un de ses arguments accepte que celui-ci ait un type non entièrement déterminé. Ce paramètre est alors dit *polymorphe*. Cette particularité permet de développer un code générique utilisable pour des structures de données différentes tant que la représentation exacte de cette structure n'a pas besoin d'être connue par le code en question. L'algorithme de typage est à même de faire cette distinction.

Objective CAML possède une inférence de types :

le programmeur n'a besoin de donner aucune information de type à l'intérieur de son programme. Le langage se charge seul de déduire du code le type le plus général des expressions et des déclarations qui y figurent. Cette *inférence* est effectuée conjointement à la

vérification, lors de la compilation du programme.

Objective CAML est muni d'un mécanisme d'exceptions :

il est possible de rompre l'exécution normale d'un programme à un endroit et de la reprendre à un autre endroit du programme prévu à cet effet. Ce mécanisme permet de gérer les situations exceptionnelles, mais il peut aussi être adopté comme style de programmation.

Objective CAML possède des traits impératifs :

les entrées-sorties, les modifications physiques de valeurs et les structures de contrôle itératives sont possibles sans avoir recours aux traits de la programmation fonctionnelle. Le mélange des deux styles est bien entendu accepté et offre une grande souplesse de développement ainsi que la possibilité de définir des structures de données nouvelles.

Objective CAML exécute des processus légers (threads) :

les principaux outils de création, de synchronisation pour la gestion de la mémoire partagée et de communication entre différents processus légers sont prédéfinis.

Objective CAML communique sur le réseau Internet :

les bases nécessaires à l'ouverture de canaux de communication entre différentes machines sont prédéfinies et permettent la réalisation d'applications suivant l'architecture client-serveur.

Objective CAML dispose de nombreuses bibliothèques :

structures de données classiques, entrées-sorties, interfaçage avec

les ressources du système, analyses lexicale et syntaxique, calcul sur les grands nombres, valeurs persistantes, etc.

Objective CAML dispose d'un environnement de programmation :

incluant une boucle d'interaction, une trace de l'exécution, un calcul des dépendances et une analyse de performance.

Objective CAML interagit avec le langage C :

via l'appel de fonctions C à partir d'un programme Objective CAML et réciproquement, permettant ainsi l'accès aux nombreuses bibliothèques C.

Objective CAML dispose de trois modes d'exécution :

interactif par le biais d'une boucle d'interaction, compilation vers du code-octet interprété par une machine virtuelle, compilation vers du code machine. Le programmeur peut donc choisir entre souplesse de développement, portabilité du code objet sur différentes architectures ou performance sur une architecture donnée.

Structuration d'un programme

La réalisation d'applications importantes oblige le programmeur ou l'équipe de développement à se poser des questions d'organisation et de structuration. En Objective CAML, on dispose de deux modèles dont les avantages et les particularités sont distincts.

Modèle des modules paramétrés :

les données et les traitements sont regroupés au sein d'une même entité à deux facettes : d'un côté le code proprement dit, de l'autre

son interface. La communication entre modules s'effectue via leur interface. La description d'un type peut être masquée en n'apparaissant pas dans l'interface du module. Ces types de données abstraits facilitent les modifications d'implantation à l'intérieur d'un module sans affecter les autres modules qui s'en servent. De plus, les modules peuvent être paramétrés par d'autres modules augmentant ainsi leur réutilisabilité.

Modèle objet :

les descriptions des traitements et des données sont regroupées dans des entités appelées *classes*; un objet est une instance (valeur) d'une classe. La communication entre objets est réalisée par $< >$, l'objet receveur détermine à l'exécution (liaison retardée) le traitement correspondant au message. En cela, la programmation objet est $< >$ par les données. La structuration d'un programme provient des relations entre classes, en particulier l'héritage permet de définir une classe par extension d'une autre. Ce modèle accepte les classes concrètes, abstraites et paramétriques. De plus, il introduit le polymorphisme d'inclusion en définissant la relation de sous-typage entre classes.

Le choix entre ces deux modèles permet une grande souplesse dans l'organisation logique d'une application et facilite sa maintenance et son évolution. Il y a dualité entre ces deux modèles. On ne peut pas augmenter les composants d'un type dans un module (pas d'extensibilité des données), mais on peut ajouter de nouveaux traitements (extensibilité des traitements) sur ces données. En objet, on peut ajouter des sous-classes à une classe (extensibilité des données) pour traiter des nouveaux cas, mais on ne peut pas ajouter de nouveaux traitements visibles de la classe ancêtre (pas d'extensibilité des traitements). Néanmoins la combinaison des deux offre de nouvelles extensibilités pour les traitements et les données.

Sûreté et efficacité de l'exécution

Objective CAML possède une excellente sûreté d'exécution de ses programmes sans sacrifier pour autant leur efficacité.

Fondamentalement le typage statique est une garantie d'absence d'erreur de type à l'exécution et apporte une information statique utile pour le compilateur sans grever les performances par des tests dynamiques de types et cela y compris pour l'extension objet. De même le récupérateur automatique de mémoire est un gage de bonne sûreté. Celui d'Objective CAML est, de plus, particulièrement efficace. Le mécanisme d'exceptions garantit que le programme ne se retrouvera pas dans un état incohérent suite à une division par zéro ou un accès en dehors des bornes d'un tableau.

Plan de l'ouvrage

Le présent ouvrage est constitué de quatre parties principales, entourées de deux chapitres et agrémenté de deux annexes, d'une bibliographie, d'un index des éléments du langage et d'un index des concepts de programmation.

Chapitre 1 :

Ce chapitre est une rapide description de l'installation de la version 2.04 du langage Objective CAML pour les systèmes les plus courants (Windows, Unix et MacOS).

Partie I : Noyau du langage

La première partie est une présentation complète des éléments de base du langage Objective CAML. Le chapitre 2 est une plongée dans le noyau fonctionnel du langage. Le chapitre 3 est le pendant du précédent et décrit la partie impérative du langage. Le chapitre 4 compare les styles fonctionnel et impératif « purs » puis présente leur utilisation conjointe. Le chapitre 5 présente la bibliothèque graphique. Le chapitre 6 expose trois applications : gestion d'une base de données simple, un interprète d'un mini-Basic et un jeu solitaire bien connu, le démineur.

Partie II : Outils de développement

La deuxième partie de l'ouvrage décrit les différents outils pour la réalisation d'applications. Le chapitre 7 compare les différents modes de compilation que sont la boucle d'interaction et les compilateurs en ligne de commande de code-octet et natif. Le chapitre 8 présente les principales bibliothèques fournies avec la distribution du langage. Le

chapitre 9 explique les mécanismes de récupération automatique de mémoire et détaille celui utilisé par Objective CAML. Le chapitre 10 explique l'utilisation des outils de mise au point et d'analyse de programmes. Le chapitre 11 s'intéresse aux outils d'analyses lexicale et syntaxique. Le chapitre 12 montre comment interfacier des programmes Objective CAML avec C. Le chapitre 13 construit une bibliothèque et une application. Cette bibliothèque offre des outils pour la construction d'interfaces graphiques. L'application est une recherche de chemins de moindre coût dans un graphe, son interface graphique utilise la bibliothèque précédente.

Partie III : Organisation d'applications

La troisième partie décrit les deux organisations d'un programme que permettent les modules et les objets. Le chapitre 14 est une présentation des modules simples et paramétrés du langage. Le chapitre 15 introduit l'extension objet d'Objective CAML. Le chapitre 16 compare ces deux types d'organisation et indique l'intérêt de leur mélange pour augmenter l'extensibilité des programmes. Le chapitre 17 décrit deux applications conséquentes : les jeux à deux joueurs qui mettent en oeuvre plusieurs modules paramétrés utilisés pour deux jeux différents, et une simulation d'un monde de robots montrant la communication entre objets.

Partie IV : Concurrence et répartition

La quatrième partie introduit les programmations concurrente et répartie en détaillant la communication entre processus, légers ou non, et sur le réseau Internet. Le chapitre 18 montre le lien direct entre le langage et les bibliothèques système, en particulier les notions de processus et de communication. Le chapitre 19 amène l'indéterminisme de la programmation concurrente en présentant les processus légers d'Objective CAML. Le chapitre 20 discute de la communication entre processus, via les prises de communication réseau (sockets), dans le modèle de mémoire répartie. Le chapitre 21 présente tout d'abord une boîte à outils pour applications client-serveur. Elle est utilisée ensuite pour étendre les robots de la partie précédente au modèle client-serveur. Enfin, nous adaptons

certains programmes déjà rencontrés sous forme de serveur HTTP.

Chapitre 22

Ce dernier chapitre fait le point sur le développement d'applications en Objective CAML et présente les applications les plus connues des langages de la famille ML.

Annexes

La première annexe explique la notion de types cycliques utilisée dans le typage des objets. La deuxième annexe décrit les évolutions du langage présentes dans la version expérimentale 2.99. Celles-ci seront probablement intégrées dans les prochaines versions d'Objective CAML (3.xx).

Chaque chapitre comprend une présentation générale du thème abordé, un plan du chapitre, les différentes sections de celui-ci, des énoncés d'exercices à réaliser, un résumé et une ultime section intitulée « Pour en savoir plus » qui indique les références bibliographiques du thème abordé.

Comment obtenir Objective CAML

Logiciels libres

Les différents logiciels utilisés dans cet ouvrage sont des logiciels « libres »¹. On les trouve soit sur le cédérom accompagnant cet ouvrage, soit en téléchargement depuis le réseau Internet. C'est le cas d'Objective CAML, développé à l'Inria.

Description du Cédérom

Le cédérom est fourni sous forme d'une arborescence de fichiers. À partir de la racine on trouve le fichier `index.html` de présentation du cédérom ainsi que les cinq répertoires catalogues suivants :

- ouvrage : racine de la version HTML de l'ouvrage augmenté des solutions des exercices;
- applis : applications décrites dans l'ouvrage;
- exercices : solutions autonomes des exercices proposés;

- distrib : ensemble des distributions fournies par l'Inria, comme décrit au paragraphe suivant;
- outils : ensemble d'outils pour le développement en Objective CAML;
- docs : documentations en ligne de la distribution et des outils.

La lecture du cédérom passe par l'ouverture du fichier index.html de la racine à partir de son navigateur préféré. Pour accéder directement à la version hypertexte de l'ouvrage, il suffit d'ouvrir le fichier ouvrage/index.html. On retrouve cette arborescence, mise à jour en tenant compte des remarques des lecteurs, sur le site de l'éditeur :

Lien

<http://www.oreilly.fr>

Téléchargement

Le téléchargement d'Objective CAML s'effectue via un navigateur HTTP à l'adresse suivante :

Lien

<http://caml.inria.fr/ocaml/distrib.html> On y trouve les distributions binaires pour Linux (Intel et PPC), pour Windows (NT, 95, 98) et pour MacOS (7, 8), ainsi que la documentation, en anglais, sous différents formats (PDF, PostScript et HTML). Par ailleurs les sources sont disponibles pour les trois systèmes. Une fois la distribution désirée copiée sur sa machine, il est temps de l'installer. Cette procédure diffère selon le système d'exploitation utilisé.

Installation

L'installation d'Objective CAML nécessite environ 10Mo d'espace libre sur son disque dur. Le logiciel se désinstalle facilement sans corrompre le système.

Installation sous Windows

Le fichier de la distribution binaire s'appelle : `ocaml-2.04-win.zip`, indiquant le numéro de version (ici 2.04) et le système d'exploitation.

Warning

Objective CAML ne fonctionne que sur les versions récentes de Windows : Windows 95, 98 et NT. Ne cherchez pas à l'installer sur Windows 3.x OS2/Warp.

1. Le fichier est en format compressé (.zip); la première opération à effectuer est de le décompresser. Pour cela, utilisez votre logiciel préféré de décompression. Vous obtenez alors un catalogue dont la racine se nomme ocaml. Vous pouvez placer ce catalogue à n'importe quel emplacement de votre disque dur. On le note par la suite.
2. Ce catalogue comprend :
 - deux sous-catalogues : bin pour les binaires et lib pour les bibliothèques;
 - deux fichiers « textes » : License.txt et Changes.txt contenant la licence d'utilisation du logiciel et les modifications par rapport aux versions antérieures;
 - une application : OCamlWin correspondant à l'application principale;
 - un fichier de configuration : Ocamlwin.ini qu'il faudra modifier (voir le point suivant);
 - deux fichiers de notes sur la version : le premier, Readme.gen, pour cette version et le second, Readme.win, pour la version sous Windows.
8. Si vous avez choisi un autre catalogue que `c:\ocaml` comme catalogue de votre racine, il est alors nécessaire de l'indiquer au fichier de configuration. Éditez-le avec Wordpad et changez la ligne définissant CmdLine qui est de la forme :

```
CmdLine=ocamlrun c:\ocaml\bin\ocaml.exe -I c:\ocaml
```

en

```
CmdLine=ocamlrun <caml-dir>\bin\ocaml.exe -I <caml-
```

Il faut remplacer les chemins de recherche des binaires et des bibliothèques par le nom du catalogue racine d'Objective CAML. Si nous avons choisi C:\Lang\ocaml comme catalogue racine (), la modification devient :

```
CmdLine=ocamlrun C:\Lang\ocaml\bin\ocaml.exe -I C:\
```

9. Copier le fichier OCamlWin.ini dans le catalogue principal du système, c'est-à-dire C:\windows ou C:\win95 ou C:\winnt selon l'installation de votre système.

Il est alors temps de tester l'application OCamlWin en double-cliquant dessus. Vous obtiendrez la fenêtre de la figure 1.1.

[image]

Figure 1.1 : Fenêtre Objective CAML sous Windows

La configuration des commandes en ligne, lancées à partir d'une fenêtre DOS, s'effectue en modifiant le chemin d'accès aux commandes (PATH) et la variable d'accès aux bibliothèques d'Objective CAML (CAMLLIB), de la manière suivante :

```
PATH=%PATH%;<caml-dir>\bin  
set CAMLLIB=<caml-dir>\lib
```

où est remplacé par le chemin où Objective CAML est installé.

Ces deux commandes peuvent être incluses dans le fichier autoexec.bat que tout bon DOS possède. Pour tester les commandes en ligne, taper dans une fenêtre DOS, la commande ocaml. Celle-ci exécute le fichier :

```
<caml-dir>/bin/ocaml.exe
```

correspondant à la boucle d'interaction en mode texte d'Objective CAML. Pour sortir de cette commande, écrire `#quit;;`.

L'installation d'Objective CAML à partir des sources sous Windows n'est pas chose aisée, car elle nécessite d'utiliser des logiciels payants, en particulier le compilateur C de Microsoft. Reportez-vous au fichier Readme.win de la distribution binaire pour avoir des précisions.

Installation sous Linux

L'installation pour Linux possède aussi une distribution binaire facile à installer sous forme paquetage rpm. L'installation à partir des sources est décrite dans la section 1. Le fichier à télécharger est : `ocaml-2.04-2.i386.rpm` que l'on emploiera de la manière suivante avec les droits de root :

```
rpm -i ocaml-2.04-2.i386.rpm
```

qui installe les commandes dans le catalogue `/usr/bin` et les bibliothèques dans le catalogue `/usr/lib/ocaml`.

Pour tester l'installation, taper : `ocamlc -v` qui affiche la version d'Objective CAML installée sur la machine.

```
ocamlc -v
The Objective Caml compiler, version 2.04
Standard library directory: /usr/lib/ocaml
```

Vous pouvez aussi exécuter la commande `ocaml` qui affiche l'entête de la boucle d'interaction.

```
Objective Caml version 2.04
```

```
#
```

Le caractère `#` est l'invite (prompt) de la boucle d'interaction. La sortie de cette boucle d'interaction s'effectue par la directive `#quit;;`, ou en tapant `CTRL-D`. Les deux points-virgule indiquent

la fin d'une phrase Objective CAML.

Installation sous MacOS

La distribution pour MacOS existe aussi sous forme d'un binaire auto-extractible. Le fichier à télécharger est : ocaml-2.04-mac.sea.bin qui est compressé. On utilisera son logiciel préféré pour le décompresser. Ensuite il suffit de lancer l'archive auto-extractible pour l'installation et suivre les instructions affichées dans la fenêtre de dialogue pour le choix de l'emplacement de la distribution.

Pour la distribution MacOS X server, suivre l'installation sous Unix à partir des sources.

Installation sous Unix à partir des sources

L'installation d'Objective CAML pour les systèmes de la famille Unix s'effectue à partir de la distribution avec sources. En effet il sera nécessaire de compiler le système Objective CAML. Pour ce faire, il faut avoir un compilateur C sur sa machine Unix, ce qui est généralement le cas, ou d'en télécharger un comme gcc qui fonctionne sur la plupart des systèmes Unix. Le fichier de la distribution d'Objective CAML contenant les sources est : ocaml-2.04.tar.gz. Le fichier INSTALL décrit, de manière fort claire, les différentes étapes de configuration, de construction puis d'installation des binaires.

Installation de la documentation HTML

La documentation en anglais d'Objective CAML existe aussi sous forme d'une arborescence de fichiers HTML que l'on retrouve dans le catalogue docs du cédérom.

Cette documentation est un manuel de référence. Il n'est pas de lecture facile pour le débutant. Néanmoins il est fort utile pour la description du langage, de ses outils et de ses bibliothèques. Il deviendra vite indispensable pour qui souhaite écrire un programme de plus de dix lignes.

Test de l'installation

Une fois l'installation de l'environnement de développement d'Objective CAML terminée, il est nécessaire de la tester pour principalement vérifier les chemins d'accès aux commandes et aux bibliothèques. Le plus simple est de lancer la boucle d'interaction du système et d'écrire le premier petit programme suivant :

```
String.concat "/" ["un"; "chemin"; "ici"] ;;
```

Cette expression concatène les différentes chaînes de caractères en insérant le caractère < > entre chaque mot. La notation `String.concat` indique l'utilisation de la fonction `concat` de la bibliothèque `String`. Si les chemins d'accès aux bibliothèques ne sont pas bons, le système affichera une erreur. On notera que le système indique que le calcul retourne une chaîne de caractères et affiche le résultat.

La documentation sur cette fonction `String.concat` peut être trouvée dans le manuel de référence en ligne en suivant les liens [The standard library](#) puis [Module String: string operations](#).

Pour sortir de la boucle d'interaction, l'utilisateur doit écrire la directive `«#quit ; ;»`.

~

Part I Noyau du langage

Présentation de la partie I

La première partie de cet ouvrage est une introduction complète au noyau du langage Objective CAML, en particulier le mécanisme d'évaluation d'une expression, le typage statique et le modèle mémoire des données.

Une expression est la description d'un calcul. L'évaluation d'une expression retourne à la fin du calcul une valeur. L'exécution d'un programme en Objective CAML correspond au calcul d'une expression. Les fonctions, les structures de contrôle de l'exécution d'un programme, telles les conditionnelles ou les boucles, sont elles aussi des expressions.

Le typage statique garantit que le calcul d'une expression ne pourra pas déclencher une erreur de type à l'exécution. Dans les faits, l'application d'une fonction à des arguments (ou paramètres effectifs) n'est acceptée que s'ils ont tous un type compatible avec les paramètres formels indiqués dans la définition de la fonction. Le langage Objective CAML possède en outre une inférence de types : le compilateur détermine automatiquement le type le plus général d'une expression.

Enfin la connaissance minimale de la représentation des données est indispensable au programmeur pour maîtriser les effets des modifications physiques sur les données.

Plan

Le chapitre 2 contient une présentation complète de la partie purement fonctionnelle du langage et des contraintes dues au typage statique. La notion de calcul d'expression y est longuement illustrée. Les structures de contrôle suivantes sont détaillées : la

conditionnelle, l'application et le filtrage de motif. Les différences entre type et domaine d'une fonction sont discutées afin d'introduire le mécanisme d'exceptions. Ce trait du langage sort du cadre fonctionnel et permet la gestion de ruptures de calcul.

Le chapitre 3 expose le style impératif. Les constructions y sont plus proches des langages classiques. Les structures de contrôle associées comme la séquence et l'itération y sont présentées ainsi que les structures de données physiquement modifiables. L'interaction entre modifications physiques et partage de données est alors détaillée. L'inférence de types y est précisée en tenant compte de ces nouvelles constructions.

Le chapitre 4 compare les deux styles précédents et surtout présente différents styles mixtes. Ce mélange permettra en particulier de construire des structures de données paresseuses, y compris physiquement modifiables.

Le chapitre 5 montre l'utilisation de la bibliothèque Graphics incluse dans la distribution du langage. Les notions de base de la programmation graphique y sont exposées et immédiatement mises en oeuvre. Il en est de même pour la construction d'interfaces graphiques grâce à la gestion d'événements minimale fournie par cette bibliothèque.

Ces quatre premiers chapitres sont illustrés par un exemple complet, l'implantation

d'une calculatrice, qui évolue de chapitre en chapitre.

Le chapitre 6 présente trois applications complètes : une petite base de données, l'interprète d'un mini-BASIC et le jeu Démineur. Les deux premiers exemples sont construits principalement dans un style fonctionnel, alors que le troisième l'est dans un style impératif.

Rudiments de syntaxe

Avant de commencer nous indiquons les premiers éléments de syntaxe du langage. Un programme est une suite de phrases du langage. Une phrase est un élément de syntaxe complet directement

exécutable (expression, déclaration). La fin d'une phrase se termine par un double point-virgule (;). On distingue trois types de déclarations qui sont repérées par un mot clé différent :

déclaration de valeur	:	let
déclaration d'exception	:	exception
déclaration de type	:	type

Tous les exemples donnés dans cette partie sont à écrire au niveau de la boucle d'interaction du langage.

Voici un premier (petit) programme Objective CAML, entré sous la boucle d'interaction, dont l'invite est le caractère dièse (#), où sont définies une fonction `fact` calculant la factorielle d'un nombre, et son application à l'entier 8.

```
# let rec fact n = if n < 2 then 1 else n * fact(n-1) ;;
val fact : int -> int = <fun>
# fact 8 ;;
- : int = 40320
```

Ce programme comprend deux *phrases*. La première est la déclaration d'une valeur fonctionnelle et la seconde est une expression. On s'aperçoit que la boucle d'interaction affiche trois informations qui sont : le nom de la déclaration, ou le caractère moins (-) dans le cas d'une expression ; le type inféré ; et la valeur retournée. Dans le cas d'une valeur fonctionnelle, le système affiche .

L'exemple suivant montre la manipulation de fonctions comme valeurs du langage. On y définit tout d'abord la fonction `succ` qui calcule le successeur d'un entier, puis la fonction `compose` qui compose deux fonctions. Celle-ci sera appliquée à `fact` et `succ`.

```
# let succ x = x+1 ;;
val succ : int -> int = <fun>
# let compose f g x = f(g x) ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# compose fact succ 8 ;;
```

```
- : int = 362880
```

Ce dernier appel effectue le calcul `fact(succ 8)` et retourne le résultat escompté. Notons que les fonctions `fact` et `succ` sont passées en paramètres à `compose` au même titre que l'entier `8`.

Programmation fonctionnelle

Introduction

Le premier langage fonctionnel, Lisp, est apparu à la fin des années 1950. C'est-à-dire au même moment que Fortran, premier représentant des langages impératifs. Ces deux langages existent toujours, bien qu'ayant fortement évolués. Ils couvrent respectivement le domaine de la programmation d'applications numériques pour Fortran et d'applications symboliques pour Lisp. L'intérêt de la programmation fonctionnelle provient d'une grande facilité d'écriture des programmes et des valeurs qu'ils manipulent. Un programme est une fonction appliquée à ses arguments. Il calcule un résultat qui est retourné (quand le calcul termine) comme sortie du programme. Il devient alors facile de composer des programmes : la sortie d'un programme devient l'argument d'entrée d'un autre, au sens de la composition de fonctions.

La programmation fonctionnelle repose sur un modèle de calcul simple possédant trois constructions : les variables, la définition d'une fonction et son application à un argument. Ce modèle s'appelle le λ -calcul et il a été introduit par Alonzo Church en 1932, donc avant le premier ordinateur. Il a été conçu pour offrir un modèle théorique général de la notion de calculabilité. En λ -calcul, toutes les fonctions sont des valeurs manipulables. Elles peuvent être utilisées comme paramètres d'autres fonctions ou être retournées comme résultat de l'appel d'une autre fonction. La théorie du λ -calcul affirme que tout ce qui est calculable (entendez programmable) peut être écrit dans ce formalisme. Cependant sa syntaxe est trop limitée pour être utilisable comme langage de programmation. En conséquence, il lui a été ajouté des valeurs de base (tels que les entiers ou les chaînes de caractères), des

opérateurs sur ces valeurs de base, des structures de contrôle et les déclarations qui permettent de nommer des valeurs ou des fonctions et, en particulier, des fonctions récursives.

Il existe plusieurs classifications des langages fonctionnels. Pour notre part, nous les distinguons suivant deux caractéristiques qui nous paraissent prépondérantes :

- Sans effets de bord (pur) ou avec effets de bord (impur) : un langage fonctionnel pur est un langage où l'affectation n'existe pas. Tout n'y est que calcul et l'on ne s'intéresse pas à son déroulement. Les langages fonctionnels impurs, comme Lisp ou ML, intègrent des traits impératifs comme l'affectation. Ils permettent d'écrire des algorithmes dans un style plus proche de langages comme Fortran où l'ordre d'évaluation des expressions est déterminant.
- Typé dynamiquement ou typé statiquement : le typage permet de vérifier si un argument passé à une fonction est bien du type du paramètre formel de la fonction. On peut faire cette vérification à l'exécution du programme. On appellera alors cette vérification le typage dynamique. En cas d'erreur sur les types le programme s'arrêtera dans un état cohérent. C'est le cas du langage Lisp. Cette vérification peut aussi se faire avant l'exécution du programme, c'est-à-dire au moment de la compilation. On appelle cette vérification a priori le typage statique. Étant effectuée une fois pour toute, elle ne ralentira pas l'exécution du programme. C'est le cas du langage ML et de ses dialectes comme Objective CAML. Seuls les programmes correctement typés, c'est-à-dire ceux acceptés par le vérificateur de types, pourront être compilés puis exécutés.

Plan du chapitre

Ce chapitre présente les éléments de base de la partie fonctionnelle du langage Objective CAML, à savoir ses éléments syntaxiques, son langage de types et son mécanisme d'exceptions. Ceci nous amènera à la réalisation d'un premier exemple de programme complet.

La première section décrit le noyau du langage en commençant par les valeurs de base et les fonctions qui les manipulent. Nous passons ensuite aux valeurs structurées et aux valeurs fonctionnelles. Les structures de contrôle de base sont introduites ainsi que les déclarations locales et globales de valeurs. La deuxième section s'intéresse aux définitions de types pour la construction de valeurs structurées et au filtrage de motifs pour l'accès à ces structures. La troisième section compare le type inféré des fonctions et leur domaine de définition, ce qui nous amène à introduire le mécanisme des exceptions. La quatrième section illustre l'ensemble de ces notions en décrivant une application simple : une calculatrice de bureau.

Noyau fonctionnel d'Objective CAML

Comme tout langage fonctionnel, Objective CAML est un langage d'expressions comprenant principalement la création de fonctions et l'application de celles-ci. Le résultat du calcul d'une de ces expressions est une valeur du langage et l'exécution d'un programme est le calcul de toutes les expressions le constituant.

Valeurs, fonctions et types de base

Les nombres entiers et flottants, les caractères, les chaînes de caractères et les booléens sont prédéfinis en Objective CAML.

Nombres

On distingue les nombres entiers¹ de type *int* et les nombres flottants de type *float*. Objective CAML suit la norme IEEE 754² pour représenter les nombres flottants en double précision. Les opérations sur les entiers et sur les flottants sont décrites à la figure 2.1. Notons que lorsque le résultat d'une opération entière sort de l'intervalle de définition des valeurs de type *int*, cela ne produit pas une erreur mais le résultat est un entier dans l'intervalle des entiers du système. En d'autres termes, toutes les opérations sur les entiers sont des opérations *modulo* les bornes de l'intervalle.

```

- : float = 5
# 3.0 + 2.0 ;;
Characters 0-3:
This expression has type float but is here used with type
# sin 3.14159 ;;
- : float = 2.65358979335e-06

```

Un calcul non défini, comme une division entière par zéro, déclenchera une exception (voir page ??) qui interrompt le calcul. Les nombres flottants ont une représentation pour les valeurs infinies (affichées comme Inf) et les calculs non définis (affichés comme NaN³). Les principales fonctions sur les nombres flottants sont décrites à la figure 2.2.

fonctions sur les flottants fonctions
trigonométriques

```

# expm1 11354078 ;;
- : float = 0.9999999999867
# sin(x as float 0.707) ;;
- : float = 0.707
# atan2 0.03; 1) ;;
- : float = 1357079632679
# floor 3(143; 1) ;;
- : float = -4                      NaN

```

Figure 2.2 : fonctions sur les flottants

Caractères et Chaînes

Les caractères, type *char*, correspondent aux entiers compris entre 0 et 255 selon le codage ASCII pour les 128 premiers. Les fonctions *char_of_int* et *int_of_char* permettent les conversions entre entiers et caractères. Les chaînes de caractères, type *string*, sont des suites de caractères de longueur connue (inférieure à $2^{24}-6$). L'opérateur de

concaténation est . Les fonctions `int_of_string`, `string_of_int`, `string_of_float` et `float_of_string` effectuent les différentes conversions entre nombres et chaînes de caractères.

```
# 'B' ;;
- : char = 'B'
# int_of_char 'B' ;;
- : int = 66
# "est une chaîne" ;;
- : string = "est une cha\238ne"
# (string_of_int 1987) ^ " est l'année de la création de "
- : string = "1987 est l'ann\233e de la cr\233ation de C"
```

Même si une chaîne contient les caractères d'un nombre, il ne sera pas possible de l'utiliser dans des opérations sur les nombres sans effectuer une conversion explicite.

```
# "1999" + 1 ;;
Characters 1-7:
This expression has type string but is here used with type int
# (int_of_string "1999") + 1 ;;
- : int = 2000
```

Les nombreuses fonctions sur les chaînes de caractères sont regroupées dans le module `String` (voir page ??).

Booléens

Un booléen appartient à un ensemble formé de deux valeurs : `true` et `false`. Les opérateurs de base sont décrits à la figure 2.3. Pour des raisons historiques, les opérateurs « et » et « ou » possèdent deux formes.

Figure 2.3 : opérateurs sur les booléens

```
# true ;;
- : bool = true
# not true ;;
- : bool = false
# true && false ;;
- : bool = false
```

Les opérateurs && et ||, ou leurs synonymes, évaluent leur argument de gauche et, selon sa valeur, évaluent leur argument à droite. Ils peuvent se réécrire sous forme de structures conditionnelles (voir page ??).

Figure 2.4 : opérateurs d'égalité et de comparaison

Les opérateurs d'égalité et de comparaison sont décrits à la figure 2.4. Ils sont polymorphes, c'est-à-dire qu'ils sont aussi bien utilisables pour comparer deux entiers que deux chaînes de caractères. La seule contrainte reste que leurs deux opérandes doivent être du même type (voir page ??).

```
# 1<=118 && (1=2 || not (1=2)) ;;
- : bool = true
# 1.0 <= 118.0 && (1.0 = 2.0 || not (1.0 = 2.0)) ;;
- : bool = true
# "un" < "deux" ;;
- : bool = false
# 0 < '0' ;;
```

Characters 4-7:

This expression has type char but is here used with type

L'égalité structurelle teste l'égalité de deux valeurs en explorant leur structure, alors que l'égalité physique teste si les deux valeurs occupent la même zone mémoire. Ces deux égalités retournent le même résultat pour les valeurs simples : booléens, caractères, entiers et constructeurs constants (page ??).

Warning

Les nombres flottants et les chaînes de caractères sont considérés comme des valeurs structurées.

Unité

Le type *unit* décrit un ensemble ne possédant qu'une seule valeur notée : ().

```
# () ;;  
- : unit = ()
```

Cette valeur sera plus particulièrement utilisée dans les programmes impératifs (3, page ??) pour les fonctions qui effectuent des effets de bord. Les fonctions dont le résultat est la valeur () simulent la notion de procédure qui n'existe pas en Objective CAML comme le fait le type void dans le langage C.

Produit cartésien, n-uplet

Des valeurs de types différents peuvent être regroupées en couple ou plus généralement en n-uplets. Les valeurs constituant un n-uplet sont séparées par une virgule. Le constructeur de type * indique un n-uplet. Le type *int* * *string* est le type des couples dont le premier élément est un entier (de type *int*) et le second une chaîne de caractères (de type *string*).

```
# ( 12 , "octobre" ) ;;  
- : int * string = 12, "octobre"
```

Lorsqu'il n'y a pas ambiguïté, on écrit plus simplement :

```
# 12 , "octobre" ;;  
- : int * string = 12, "octobre"
```

Les fonctions `fst` et `snd` permettent d'accéder au premier et au second élément d'un couple.

```
# fst ( 12 , "octobre" ) ;;
- : int = 12
# snd ( 12 , "octobre" ) ;;
- : string = "octobre"
```

Ces deux fonctions acceptent des couples dont les composants sont de type quelconque. Elles sont polymorphes, à l'instar de l'égalité.

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# fst ( "octobre", 12 ) ;;
- : string = "octobre"
```

Le type `int * char * string` est celui des triplets dont le premier élément est de type `int`, le deuxième de type `char` et le troisième de type `string`. Ses valeurs s'écrivent

```
# ( 65 , 'B' , "ascii" ) ;;
- : int * char * string = 65, 'B', "ascii"
```

Warning

Les fonctions `fst` et `snd` appliquées à un n-uplet, autre qu'un couple, provoquent une erreur de type.

```
# snd ( 65 , 'B' , "ascii" ) ;;
```

Characters 7-25:

This expression has type `int * char * string` but is here
'a * 'b

Il y a effectivement une différence entre le type d'un couple et celui d'un triplet. Le type `int * int * int` est différent des types `(int * int) * int` et `int * (int * int)`. Les accesseurs d'un triplet (et des autres n-uplets) ne sont pas définis dans la bibliothèque de base. On utilisera

le filtrage de motifs pour les définir si besoin est (voir page [??](#)).

Listes

Des valeurs d'un même type peuvent être regroupées en liste. Une liste peut être soit vide soit constituée d'éléments du même type.

```
# [] ;;
- : 'a list = []
# [ 1 ; 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; "deux" ; 3 ] ;;
```

Characters 15-18:

This expression has type `int list` but is here used with

La fonction qui ajoute un élément en tête de liste est l'opérateur infixe `::`. C'est l'analogue du *cons* de Lisp.

```
# 1 :: 2 :: 3 :: [] ;;
- : int list = [1; 2; 3]
```

La concaténation de deux listes est aussi un opérateur infixe `@`.

```
# [ 1 ] @ [ 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; 2 ] @ [ 3 ] ;;
- : int list = [1; 2; 3]
```

Les autres fonctions de manipulation des listes sont définies dans la bibliothèque `List`. Les fonctions `hd` et `tl` de cette bibliothèque donnent respectivement la tête et la queue d'une liste quand ces valeurs existent. On note ces fonctions `List.hd` et `List.tl` pour indiquer au système qu'il doit les trouver dans le module `List`⁴.

```
# List.hd [ 1 ; 2 ; 3 ] ;;
- : int = 1
# List.hd [] ;;
Uncaught exception: Failure("hd")
```

Dans ce dernier exemple, il est effectivement problématique de vouloir récupérer le premier élément d'une liste vide. C'est pour cela que le système déclenche une exception (voir page ??).

Structure de contrôle conditionnelle

Une des structures de contrôle indispensable pour tout langage de programmation est la structure dite conditionnelle (ou alternative) qui oriente le calcul en fonction d'une condition.

Syntaxe

if*expr1***then***expr2***else***expr3* L'expression *expr1* est de type *bool*. Les expressions *expr2* et *expr3* doivent être du même type, quel qu'il soit.

```
# if 3=4 then 0 else 4 ;;
- : int = 4
# if 3=4 then "0" else "4" ;;
- : string = "4"
# if 3=4 then 0 else "4";;
```

Characters 20-23:

This expression has type string but is here used with type

Une expression conditionnelle est, elle aussi, une expression et son calcul retourne une valeur.

```
# (if 3=5 then 8 else 10) + 5 ;;
- : int = 15
```

Remarque

La branche **else** peut être omise, mais, dans ce cas elle est implicitement remplacée par **else ()**. En conséquence le type de l'expression *expr2* doit être *unit* (voir page ??).

Déclarations de valeurs

Une déclaration associe un nom à une valeur. On distingue les déclarations globales des déclarations locales. Dans le premier cas, les noms déclarés sont connus de toutes les expressions suivant la déclaration ; dans le second, les noms déclarés ne sont connus que d'une expression. Il est également possible de déclarer *simultanément* plusieurs associations nom-valeur.

Déclarations globales

Syntaxe

let*nom* = *expr*;; Une déclaration globale définit l'association du nom *nom* à la valeur de l'expression *expr* qui sera connue de toutes les expressions ultérieures.

```
# let an = "1999" ;;
val an : string = "1999"
# let x = int_of_string(an) ;;
val x : int = 1999
# x ;;
- : int = 1999
# x + 1 ;;
- : int = 2000
# let nouvel_an = string_of_int (x + 1) ;;
val nouvel_an : string = "2000"
```

Déclarations globales simultanées

Syntaxe

```
let nom1 = expr1
and nom2 = expr2
:
```


and *nomn* = *exprn* ;;

Une déclaration simultanée déclare au même niveau différents symboles. Ils ne seront connus qu'à la fin de toutes les déclarations.

```
# let x = 1 and y = 2 ;;
val x : int = 1
val y : int = 2
# x + y ;;
- : int = 3
# let z = 3 and t = z + 2 ;;
Characters 18-19:
Unbound value z
```

Il est possible de regrouper plusieurs déclarations globales dans une même phrase, l'affichage de leur type et de leur valeur n'intervient alors qu'à la fin de la phrase marquée par le double « ;; ». Ces déclarations sont évaluées séquentiellement à la différence d'une déclaration combinée.

```
# let x = 2
  let y = x + 3 ;;
val x : int = 2
val y : int = 5
```

Une déclaration globale peut être masquée par une nouvelle déclaration du même nom (voir page [??](#)).

Déclarations locales

Syntaxe

let*nom* = *expr1* **in** *expr2*;; Le nom *nom* n'est connu que pour le calcul de *expr2*. La déclaration locale lui associe la valeur de *expr1*.

```
# let x1 = 3 in x1 * x1 ;;
```

```
- : int = 9
```

La déclaration locale liant `xl` à la valeur 3 n'est conservée que pour le temps du calcul de `xl * xl`.

```
# xl ;;
```

```
Characters 1-3:
```

```
Unbound value xl
```

Une déclaration locale masque toute déclaration antérieure du même nom, mais l'ancienne valeur est retrouvée lorsque l'on sort de la portée de la déclaration locale :

```
# let x = 2 ;;
```

```
val x : int = 2
```

```
# let x = 3 in x * x ;;
```

```
- : int = 9
```

```
# x * x ;;
```

```
- : int = 4
```

Une déclaration locale est une expression et peut donc être utilisée pour construire d'autres expressions :

```
# (let x = 3 in x * x) + 1 ;;
```

```
- : int = 10
```

Les déclarations locales peuvent aussi être simultanées.

Syntaxe

let

nom1 = expr1

and

nom2 = expr2

:

and

nomn = exprn

in

expr ;;

```
# let a = 3.0 and b = 4.0 in sqrt (a*.a +. b*.b) ;;
```

```
- : float = 5
# b ;;
Characters 0-1:
Unbound value b
```

Expressions fonctionnelles, fonctions

Une expression fonctionnelle est constituée d'un *paramètre* et d'un *corps*. Le paramètre formel est un nom de variable et le corps une expression. On dit que le paramètre est abstrait. C'est pour cela qu'une expression fonctionnelle est aussi appelée abstraction.

Syntaxe

function*p* -> *expr*

Ainsi la fonction qui élève au carré son argument s'écrit :

```
# function x -> x*x ;;
- : int -> int = <fun>
```

Le système Objective CAML déduit son type. Le type fonctionnel *int* -> *int* indique une fonction attendant un paramètre de type *int* et retournant une valeur de type *int*. L'application d'une fonction à un argument s'écrit comme la fonction suivie par l'argument.

```
# (function x -> x * x) 5 ;;
- : int = 25
```

Le calcul d'une application revient à calculer le corps de la fonction, ici $x * x$, où le paramètre formel, x , est remplacé par la valeur de l'argument (ou paramètre d'appel, ou encore paramètre effectif), ici 5.

Dans la construction d'une expression fonctionnelle, *expr* est une

expression quelconque. En particulier, `expr` peut elle-même être une expression fonctionnelle.

```
# function x -> (function y -> 3*x + y) ;;  
- : int -> int -> int = <fun>
```

Les parenthèses ne sont pas obligatoires. On peut écrire plus simplement :

```
# function x -> function y -> 3*x + y ;;  
- : int -> int -> int = <fun>
```

Le type de cette expression peut être lu de la façon usuelle comme le type d'une fonction qui attend deux entiers et retourne une valeur entière. Mais dans le cadre d'un langage fonctionnel comme Objective CAML il s'agit plus exactement du type d'une fonction qui attend un entier et retourne une valeur fonctionnelle de type `int -> int` :

```
# (function x -> function y -> 3*x + y) 5 ;;  
- : int -> int = <fun>
```

On peut, bien entendu, utiliser l'expression fonctionnelle de façon usuelle en l'appliquant à deux arguments. On écrit :

```
# (function x -> function y -> 3*x + y) 4 5 ;;  
- : int = 17
```

Lorsque l'on écrit `f a b`, il y a un parenthésage implicite à gauche ce qui rend cette expression équivalente à : `(f a) b`.

Revenons sur le détail de l'application

```
(function x -> function y -> 3*x + y) 4 5
```

Pour calculer la valeur de cette expression, il faut calculer la valeur de

```
(function x -> function y -> 3*x + y) 4
```

qui est une *expression fonctionnelle* équivalente à

```
function y -> 3*4 + y
```

obtenue en remplaçant x par 4 dans $3*x + y$. En appliquant cette valeur (qui est une fonction) à 5 on obtient la valeur finale $3*4+5 = 17$:

```
# (function x -> function y -> 3*x + y) 4 5 ;;  
- : int = 17
```

Arité d'une fonction

On appelle arité d'une fonction le nombre de ces arguments. L'usage, hérité des mathématiques veut que les arguments d'une fonction soient donnés entre parenthèse après le nom de la fonction. On écrit : $f(4,5)$. Nous venons de voir qu'en Objective CAML, on écrit plutôt : $f\ 4\ 5$. On peut, bien entendu, en Objective CAML écrire une expression fonctionnelle que l'on applique à (4,5) :

```
# function (x,y) -> 3*x + y ;;  
- : int * int -> int = <fun>
```

Mais, comme l'indique son type, cette dernière expression, n'attend pas deux, mais un seul argument : un couple d'entiers. Tenter d'appliquer deux argument à une fonction qui attend un couple ou tenter d'appliquer un couple à une fonction qui attend deux arguments provoquent une erreur de typage :

```
# (function (x,y) -> 3*x + y) 4 5 ;;
```

Characters 29-30:

This expression has type int but is here used with type

```
# (function x -> function y -> 3*x + y) (4, 5) ;;
```

Characters 39-43:

This expression has type int * int but is here used with

Syntaxe alternative

Il existe une manière plus compacte d'écrire des expressions fonctionnelles à plusieurs paramètres. C'est une survivance des anciennes versions du langage Caml. Sa forme est la suivante :

Syntaxe

fun*p*₁ ...*p*_{*n*} -> expr

Elle permet de ne pas répéter le mot clé **function** ni les flèches. Elle est équivalente à la traduction suivante :

function *p*₁ -> -> **function** *p*_{*n*} -> expr

```
# fun x y -> 3*x + y ;;  
- : int -> int -> int = <fun>  
# (fun x y -> 3*x + y) 4 5 ;;  
- : int = 17
```

On la rencontre encore souvent, en particulier dans les bibliothèques fournies avec la distribution d'Objective CAML.

Fermeture

Objective CAML considère une expression fonctionnelle comme toute autre expression et sait calculer sa valeur. La valeur retournée par le calcul d'une expression fonctionnelle est appelée une fermeture. Toute expression Objective CAML est évaluée dans un environnement constitué des associations noms-valeurs provenant des déclarations précédant l'expression à calculer. On peut décrire une fermeture comme le triplet constitué du nom du paramètre formel, du corps de la fonction et de l'environnement de l'expression. On a besoin de conserver cet environnement car le corps d'une expression fonctionnelle peut utiliser, en plus des paramètres formels, toute autre variable déclarée précédemment. Ces variables sont dites libres dans l'expression fonctionnelle. On aura besoin de leur valeur lorsque l'expression fonctionnelle sera appliquée.

```
# let m = 3 ;;
val m : int = 3
# function x -> x + m ;;
- : int -> int = <fun>
# (function x -> x + m) 5 ;;
- : int = 8
```

Lorsque l'application d'une fermeture à un argument retourne une nouvelle fermeture, celle-ci possède dans son environnement toutes les associations nécessaires à une future application. Le paragraphe sur la portée des variables (voir page ??) détaille cette notion. Nous reviendrons sur le représentation mémoire d'une fermeture au chapitre 4 (page ??) ainsi qu'au chapitre 12 (page ??).

Les expressions fonctionnelles utilisées jusqu'à présent sont *anonymes*. Il est assez pratique de pouvoir les nommer.

Déclarations de valeurs fonctionnelles

On déclare une valeur fonctionnelle de la même manière que les autres valeurs du langage par la construction **let**.

```
# let succ = function x -> x + 1 ;;
val succ : int -> int = <fun>
# succ 420 ;;
- : int = 421
# let g = function x -> function y -> 2*x + 3*y ;;
val g : int -> int -> int = <fun>
# g 1 2;;
- : int = 8
```

Pour simplifier l'écriture la notation suivante est acceptée :

Syntaxe

let*nom**p*₁ ... *p*_{*n*} = *expr* qui est équivalente à la forme suivante :

```
let nom = function p1 -> -> function pn -> expr
```

Les déclarations suivantes de succ et de g sont équivalentes à leur déclaration précédente.

```
# let succ x = x + 1 ;;
val succ : int -> int = <fun>
# let g x y = 2*x + 3*y ;;
val g : int -> int -> int = <fun>
```

On découvre le caractère pleinement fonctionnel d'Objective CAML dans l'exemple suivant où la fonction h1 est obtenue par application de g à un seul entier. On parle alors d'application partielle :

```
# let h1 = g 1 ;;
val h1 : int -> int = <fun>
# h1 2 ;;
- : int = 8
```

On peut aussi, à partir de g, définir une fonction h2 en fixant la valeur du second paramètre (y) de g :

```
# let h2 = function x -> g x 2 ;;
val h2 : int -> int = <fun>
# h2 1 ;;
- : int = 8
```

Déclaration de fonctions infixes

Certaines fonctions à deux arguments peuvent être appliquées sous forme infixe. C'est le cas de l'addition sur les entiers. On écrit $3 + 5$ pour l'application de $+$ à 3 et 5. Pour utiliser le symbole $+$ comme valeur fonctionnelle classique, il faut syntaxiquement l'indiquer en entourant ce symbole infixe par des parenthèses. La syntaxe est la suivante :

Syntaxe

(op)

L'exemple suivant définit la fonction succ en utilisant (+).


```
# (+) ;;
- : int -> int -> int = <fun>
# let succ = (+) 1 ;;
val succ : int -> int = <fun>
# succ 3 ;;
- : int = 4
```

Il est aussi possible de définir de nouveaux opérateurs. On définit un opérateur ++ d'addition de couples d'entiers

```
# let (++) c1 c2 = (fst c1)+(fst c2), (snd c1)+(snd c2)
val ++ : int * int -> int * int -> int * int = <fun>
# let c = (2,3) ;;
val c : int * int = 2, 3
# c ++ c ;;
- : int * int = 4, 6
```

Il y a une limitation importante sur les opérateurs possibles. Ils ne doivent contenir que des *symboles* (tels *, +, \$, etc.) à l'exclusion des lettres et des chiffres. Font exception à la règle certaines fonctions prédéfinies comme infixes dont la liste suit : or mod land lor lxor lsl lsr asr.

Fonctions d'ordre supérieur

Une valeur fonctionnelle (une fermeture) peut être retournée comme résultat. Elle peut également être prise comme argument d'une fonction. Les fonctions prenant en argument ou retournant des valeurs fonctionnelles sont dites d'ordre supérieur.

```
# let h = function f -> function y -> (f y) + y ;;
val h : (int -> int) -> int -> int = <fun>
```

Remarque

L'application est parenthésée à gauche, mais les types fonctionnels sont parenthésés à droite. Ainsi le type de la fonction h peut s'écrire

$(int \rightarrow int) \rightarrow int \rightarrow int$ ou $(int \rightarrow int) \rightarrow (int \rightarrow int)$

Les fonctions d'ordre supérieur offrent une possibilité élégante de traiter les listes. Par exemple la fonction `List.map` permet d'appliquer une fonction à tous les éléments d'une liste et de retourner la liste des résultats.

```
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let square x = string_of_int (x*x) ;;
val square : int -> string = <fun>
# List.map square [1; 2; 3; 4] ;;
- : string list = ["1"; "4"; "9"; "16"]
```

Autre exemple, la fonction `List.for_all` permet de savoir si tous les éléments d'une liste satisfont un certain critère.

```
# List.for_all ;;
- : ('a -> bool) -> 'a list -> bool = <fun>
# List.for_all (function n -> n<>0) [-3; -2; -1; 1; 2; 3]
- : bool = true
# List.for_all (function n -> n<>0) [-3; -2; 0; 1; 2; 3]
- : bool = false
```

Portée des variables

Pour que l'évaluation d'une expression soit possible, il faut que toutes les variables qui y apparaissent soient définies. C'est en particulier le cas pour l'expression `e` de la déclaration `let p = e`. Or `p` n'étant pas encore connue dans cette expression, cette variable ne peut être présente que si elle référence une autre valeur issue d'une déclaration antérieure.

```
# let p = p ^ "-suffixe" ;;
Characters 9-10:
Unbound value p
# let p = "préfixe" ;;
val p : string = "pr\233fixe"
# let p = p ^ "-suffixe" ;;
val p : string = "pr\233fixe-suffixe"
```

En Objective CAML, les variables sont liées statiquement.

L'environnement utilisé pour exécuter l'application d'une fermeture est celui de sa déclaration (portée statique) et non celui courant au moment de l'application (portée dynamique).

```
# let p = 10 ;;
val p : int = 10
# let k x = (x, p, x+p) ;;
val k : int -> int * int * int = <fun>
# k p ;;
- : int * int * int = 10, 10, 20
# let p = 1000 ;;
val p : int = 1000
# k p ;;
- : int * int * int = 1000, 10, 1010
```

La fonction *k* possède une variable libre : *p*. Celle-ci étant définie dans l'environnement global, la définition de *k* est acceptée. La liaison entre le nom *p* et la valeur 10 dans l'environnement de la fermeture *k* est statique, c'est-à-dire ne dépend pas de la dernière définition de *p*.

Déclarations récursives

Une déclaration de variable est dite récursive si elle utilise son propre identificateur dans sa définition. Cette possibilité est principalement utilisée pour les fonctions, notamment pour simuler une définition par récurrence. Nous venons de voir que la déclaration **let** ne le permet pas. Pour déclarer une fonction récursive on utilisera une construction syntaxique dédiée.

Syntaxe

let rec*nom* = *expr*;; On peut également utiliser la facilité syntaxique des définitions de valeurs fonctionnelles en indiquant les paramètres des fonctions :

Syntaxe

let rec *comp1 ...pn = expr*;; En guise d'exemple, voici la fonction **sigma** qui calcule la somme des entiers (positifs) compris entre 0 et la valeur de son argument.

```
# let rec sigma x = if x = 0 then 0 else x + sigma (x-1)
val sigma : int -> int = <fun>
# sigma 10 ;;
- : int = 55
```

On peut remarquer que cette fonction risque de ne pas terminer si son argument est strictement négatif.

Une valeur récursive est en général une fonction. Et le compilateur rejette certaines déclarations récursives dont la valeur n'est pas fonctionnelle :

```
# let rec x = x + 1 ;;
Characters 13-18:
This kind of expression is not allowed as right-hand side of
```

Nous verrons cependant que dans certains cas de telles déclarations sont admises (voir page ??). La déclaration **let rec** peut être combinée avec la construction **and**. Dans ce cas, toutes les fonctions définies au même niveau sont connues dans le corps de toutes les autres. Cela permet, entre autres, des déclarations de fonctions mutuellement récursives.

```
# let rec pair    n = (n<>1) && ((n=0) or (impair (n-1)))
  and impair n = (n<>0) && ((n=1) or (pair (n-1)))      ;;
val pair : int -> bool = <fun>
val impair : int -> bool = <fun>
# pair 4 ;;
- : bool = true
# impair 5 ;;
- : bool = true
```

De même, les déclarations locales peuvent être récursives. Cette nouvelle définition de **sigma** teste la validité de l'argument avant d'effectuer le calcul de la somme définie par une fonction locale

sigma_rec.

```
# let sigma x =  
  let rec sigma_rec x = if x = 0 then 0 else x + sigma_re  
  if (x<0) then "erreur : argument négatif"  
  else "sigma = " ^ (string_of_int (sigma_rec x)) ;;  
val sigma : int -> string = <fun>
```

Remarque

La nécessité de donner une valeur de retour qui soit de même type, que l'argument soit négatif ou non, nous a poussé à rendre le résultat sous forme d'une chaîne de caractères. Car quelle valeur doit rendre sigma quand son argument est négatif ? Nous verrons la bonne façon de régler ce problème en utilisant les exceptions (voir page ??).

Polymorphisme et contrainte de type

Un certain nombre de fonctions exécutent le même code pour des arguments ayant des types différents. Par exemple, la création d'un couple à partir de deux valeurs ne nécessite pas d'avoir des fonctions différentes pour chaque type connu du système⁵. De même, l'accès au premier champ d'un couple n'a pas à être différencié selon le type de la valeur de ce premier champ.

```
# let make_pair a b = (a,b) ;;  
val make_pair : 'a -> 'b -> 'a * 'b = <fun>  
# let p = make_pair "papier" 451 ;;  
val p : string * int = "papier", 451  
# let a = make_pair 'B' 65 ;;  
val a : char * int = 'B', 65  
# fst p ;;  
- : string = "papier"  
# fst a ;;  
- : char = 'B'
```

Les fonctions dont l'un des paramètres ou la valeur de retour est d'un type qu'il n'est pas nécessaire de préciser sont dites

polymorphes. Le synthétiseur de types contenu dans le compilateur d'Objective CAML trouve le type le plus général pour chaque expression. Dans ce cas, Objective CAML utilise des variables, ici *'a* et *'b*, pour désigner ces types généraux. Ces variables sont instanciées par le type de l'argument lors de l'application de la fonction.

Avec les fonctions polymorphes d'Objective CAML nous cumulons les avantages de pouvoir écrire un code générique utilisable pour des valeurs de tout type, tout en conservant la sûreté d'exécution du typage statique. En effet, bien que `make_pair` soit polymorphe, la valeur créée par `(make_pair 'B' 65)` possède un type bien précis qui est différent de celui de `(make_pair "papier" 451)`. En outre, la vérification des types est effectuée à la compilation, donc la généricité du code ne nuit pas à l'efficacité du programme.

Exemples de fonctions et valeurs polymorphes

Les exemples suivants de fonctions polymorphes possèdent des paramètres fonctionnels dont le type est paramétré.

La fonction `app` applique une fonction à un argument.

```
# let app = function f -> function x -> f x ;;  
val app : ('a -> 'b) -> 'a -> 'b = <fun>
```

On peut donc l'appliquer à la fonction `impair` définie précédemment :

```
# app impair 2;;  
- : bool = false
```

La fonction identité (`id`) prend un paramètre et le retourne tel quel.

```
# let id x = x ;;  
val id : 'a -> 'a = <fun>  
# app id 1 ;;  
- : int = 1
```

La fonction `compose` prend deux fonctions et une autre valeur pour composer ces l'application de ces deux fonctions sur cette valeur.

```
# let compose f g x = f (g x) ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fu
# let add1 x = x+1 and mul5 x = x*5 in compose mul5 add
- : int = 50
```

On voit que le résultat de `g` doit être de même type que le paramètre de `f`.

Les valeurs non fonctionnelles peuvent elles aussi être polymorphes. C'est le cas par exemple de la liste vide :

```
# let l = [] ;;
val l : 'a list = []
```

L'exemple suivant montre que la synthèse du type provient bien de la résolution des contraintes d'une application et non de la valeur obtenue à l'exécution.

```
# let q = List.tl [2] ;;
val q : int list = []
```

Le type de `List.tl` est `'a list -> 'a list`, donc cette fonction appliquée à une liste d'entiers retourne une liste d'entiers. Le fait qu'à l'exécution ce soit la liste vide qui est obtenue ne change rien à son type. Objective CAML engendre des types paramétrés pour toute fonction qui n'utilise pas la forme de ses arguments. Ce polymorphisme est appelé paramétrique⁶.

Contrainte de type

Comme le synthétiseur de types en Caml engendre le type le plus général, il peut être utile ou nécessaire de préciser le type d'une expression.

La forme syntaxique d'une contrainte de type est la suivante :

Syntaxe

(*expr:t*) Lorsqu'il rencontre une telle contrainte, le synthétiseur de type en tiendra compte pour la construction du type de l'expression. L'usage des contraintes de type permet :

- de rendre visible le type des paramètres d'une fonction ;
- d'interdire l'utilisation d'une fonction en dehors de son cadre ;
- de spécifier le type d'une expression, ce qui sera particulièrement pratique pour les valeurs physiquement modifiables (voir page ??).

Les exemples suivants montrent l'utilisation de telles contraintes de type

```
# let add (x:int) (y:int) = x + y ;;
val add : int -> int -> int = <fun>
# let make_pair_int (x:int) (y:int) = x,y;;
val make_pair_int : int -> int -> int * int = <fun>
# let compose_fn_int (f : int -> int) (g : int -> int) (
  compose f g x);;
val compose_fn_int : (int -> int) -> (int -> int) -> int
# let nil = ([] : string list);;
val nil : string list = []
# 'H'::nil;;
```

Characters 5-8:

This expression has type string list but is here used wi

Cette restriction du polymorphisme permet de mieux contrôler le type des expressions en restreignant le polymorphisme du type déduit par le système. On peut utiliser n'importe quel type défini, pouvant contenir des variables de type, comme le montre l'exemple suivant :

```
# let llnil = ([] : 'a list list) ;;
val llnil : 'a list list = []
# [1;2;3]:: llnil ;;
- : int list list = [[1; 2; 3]]
```


Le symbole `llnil` est une liste de listes de n'importe quel type.

Il s'agit ici de contraintes et non d'un typage explicite remplaçant la synthèse de type d'Objective CAML. En particulier, on ne peut pas généraliser le type plus que ne le permet l'inférence.

```
# let add_general (x:'a) (y:'b) = add x y ;;  
val add_general : int -> int -> int = <fun>
```

Les contraintes de type seront utilisées dans les interfaces de modules (14) ainsi que dans les déclarations de classes (15).

Exemples

Nous donnons dans ce paragraphe quelques exemples un peu élaborés de fonctions. La plupart de ces fonctions sont prédéfinies en Objective CAML. Nous les redéfinissons à titre « pédagogique ».

Ici, le test du cas d'arrêt des fonctions récursives est réalisé par une conditionnelle. D'où un style de programmation plus proche de Lisp. Nous verrons comment donner un caractère plus ML à ces définitions lorsque nous présenterons une autre façon de définir des fonctions par cas (voir page ??).

Longueur d'une liste

Commençons par la fonction `null` qui teste si une liste est vide.

```
# let null l = (l = []) ;;  
val null : 'a list -> bool = <fun>
```

On définit alors la fonction `size` de calcul de la longueur d'une liste (i.e. le nombre de ses éléments).

```
# let rec size l =  
  if null l then 0  
  else 1 + (size (List.tl l)) ;;  
val size : 'a list -> int = <fun>
```

```
# size [] ;;
- : int = 0
# size [1;2;18;22] ;;
- : int = 4
```

La fonction `size` teste si la liste argument est vide, si oui elle retourne 0, sinon elle retourne 1 ajouté à la valeur résultant du calcul de la longueur de la queue de la liste.

Itération de composition

L'expression `iterate n f` calcule la valeur f^n qui correspond à l'application de `f` itérée n fois.

```
# let rec iterate n f =
  if n = 0 then (function x -> x)
  else compose f (iterate (n-1) f) ;;
val iterate : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

La fonction `iterate` teste si `n` vaut 0, si oui elle retourne la fonction identité, sinon elle compose `f` avec l'itération de f^{n-1} fois.

On peut, en utilisant `iterate`, définir l'élévation à la puissance comme itération de la multiplication.

```
# let rec power i n =
  let i_times = ( * ) i in
  iterate n i_times 1 ;;
val power : int -> int -> int = <fun>
# power 2 8 ;;
- : int = 256
```

La fonction `power` itère n fois l'expression fonctionnelle `i_times`, puis applique ce résultat à 1, ce qui calcule bien la puissance n -ième d'un nombre entier.

Table de multiplication

On veut écrire la fonction `multab` qui calcule la table de multiplication d'un entier passé en argument.

On définit dans un premier temps la fonction `apply_fun_list` telle que, si `f_list` est une liste de fonctions, `apply_fun_list x f_list` retourne la liste des résultats de l'application de chaque élément de `f_list` à `x`.

```
# let rec apply_fun_list x f_list =  
  if null f_list then []  
  else ((List.hd f_list) x)::(apply_fun_list x (List.tl f_list))  
val apply_fun_list : 'a -> ('a -> 'b) list -> 'b list =  
# apply_fun_list 1 [(+) 1; (+) 2; (+) 3] ;;  
- : int list = [2; 3; 4]
```

La fonction `mk_mult_fun_list` retourne la liste des fonctions multipliant leur argument par `i`, pour `i` variant de 0 à `n`.

```
# let mk_mult_fun_list n =  
  let rec mmfl_aux p =  
    if p = n then [ ( * ) n ]  
    else (( * ) p) :: (mmfl_aux (p+1))  
  in (mmfl_aux 1) ;;  
val mk_mult_fun_list : int -> (int -> int) list = <fun>
```

On obtient la table de multiplication de 7 par :

```
# let multab n = apply_fun_list n (mk_mult_fun_list 10)  
val multab : int -> int list = <fun>  
# multab 7 ;;  
- : int list = [7; 14; 21; 28; 35; 42; 49; 56; 63; 70]
```

Itération sur les listes

L'appel de la fonction `fold_left f a [e1; e2; ... ; en]` retourne `f ... (f (f a e1) e2) ... en`. Il y a donc `n` applications.

```
# let rec fold_left f a l =  
  if null l then a
```

```
else fold_left f ( f a (List.hd l)) (List.tl l) ;;  
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

La fonction `fold_left` permet la définition compacte d'une fonction de calcul de la somme des éléments d'une liste d'entiers :

```
# let sum_list = fold_left (+) 0 ;;  
val sum_list : int list -> int = <fun>  
# sum_list [2;4;7] ;;  
- : int = 13
```

Ou encore, la concaténation des éléments d'une liste de chaînes :

```
# let concat_list = fold_left (^) "" ;;  
val concat_list : string list -> string = <fun>  
# concat_list ["Hello "; "world" ; " "; "!"] ;;  
- : string = "Hello world !"
```

Déclaration de types et filtrage de motif

Bien que les types prédéfinis d'Objective CAML permettent de construire des données structurées, grâce aux n-uplets et aux listes, il est nécessaire de pouvoir définir de nouveaux types pour décrire des données structurées particulières. En Objective CAML, les déclarations de type sont récursives et peuvent être paramétrées par des variables de type, à la manière du type *'a list* déjà rencontré.

L'inférence de types tient compte de ces nouvelles déclarations pour produire le type d'une expression. La construction de valeurs de ces nouveaux types utilise les constructeurs décrits dans leur définition. Une spécificité des langages de la famille ML est le filtrage de motif. Il permet un accès simple aux composants de structures de données complexes. La définition d'une fonction correspond le plus souvent à un filtrage par motif sur un de ses paramètres permettant une définition par cas de la fonction.

Nous présentons tout d'abord le filtrage de motif sur les types prédéfinis, pour ensuite décrire les différentes déclarations de types structurés, la construction de telles valeurs ainsi que l'accès à leurs composants par filtrage de motif.

Filtrage de motif (pattern matching)

Un motif, pattern en anglais, n'est pas à proprement parler une expression d'Objective CAML. Il s'agit plutôt d'un assemblage correct (syntactiquement, et du point de vue des types) d'éléments tels que les constantes des types de base (*int*, *bool*, *char*, ..), les variables, les constructeurs et le symbole `_` dit motif universel. D'autres symboles servent à l'écriture de motifs. Nous les introduirons au fur et à mesure des besoins.

Le filtrage par motif s'applique aux valeurs. Il sert à reconnaître la forme de cette valeur et permet d'orienter le calcul en conséquence en associant à chaque motif une expression à calculer.

Syntaxe

```
match expr with  
| p1 -> expr1  
:  
| pn -> exprn
```

L'expression *expr* est filtrée séquentiellement par les différents motifs *p*₁, ..., *p*_{*n*}. Si un des motifs (par exemple *p*_{*i*}) est cohérent par rapport à la valeur de *expr* alors la branche de calcul correspondante (*expr*_{*i*}) est évaluée. Les différents motifs *p*_{*i*} sont du même type. Il en est de même pour les différentes expressions *expr*_{*i*}. La barre verticale précédant le premier motif est optionnelle.

Exemples

Voici deux façons de définir, par filtrage de motif, une fonction `imply` de type $(bool * bool) \rightarrow bool$ implantant l'implication logique. Un motif pour les couples a la forme `( , )`.

La première version de `imply` énumère tous les cas possibles comme le ferait une table de vérité :

```
# let imply v = match v with
```

```

(true,true)    -> true
| (true,false) -> false
| (false,true) -> true
| (false,false) -> true;;
val imply : bool * bool -> bool = <fun>

```

En utilisant des variables regroupant plusieurs cas, on obtient une définition plus compacte.

```

# let imply v = match v with
  (true,x)    -> x
| (false,x)   -> true;;
val imply : bool * bool -> bool = <fun>

```

Ces deux versions de `imply` calculent la même fonction. C'est-à-dire qu'elles retournent les mêmes valeurs pour les mêmes entrées.

Motif linéaire

Un motif doit être obligatoirement *linéaire*, c'est-à-dire qu'une variable donnée ne peut y figurer au plus qu'une fois. Ainsi, on aurait pu espérer pouvoir écrire :

```

# let equal c = match c with
  (x,x) -> true
| (x,y) -> false;;

```

Characters 35-36:

This variable is bound several times in this matching

Mais cela aurait exigé du compilateur de savoir réaliser les tests d'égalité. Or, la réalisation de tels tests soulève immédiatement de nombreux problèmes. Si l'on se contente de l'égalité physique entre valeurs, on obtient un système trop faible, incapable de reconnaître l'égalité entre deux occurrences de la liste `[1; 2]`, par exemple. Si l'on décide d'utiliser une égalité structurelle, on court le risque d'avoir à explorer, infiniment, des structures circulaires. Les fonctions récursives, par exemple sont des structures circulaires, mais on peut construire également des valeurs récursives, donc

circulaires, non fonctionnelles (voir page ??).

Motif universel

Le symbole `_` filtre toutes les valeurs possibles. Il est appelé motif universel. Il peut être utilisé pour filtrer des types complexes. On l'utilise, par exemple, pour simplifier encore la définition de la fonction `impl` :

```
# let impl v = match v with
  (true, false) -> false
  | _           -> true;;
val impl : bool * bool -> bool = <fun>
```

Une définition par filtrage doit considérer l'ensemble des cas possibles des valeurs filtrées. Si tel n'est pas le cas, le compilateur affiche un message d'avertissement :

```
# let is_zero n = match n with 0 -> true ;;
Characters 17-40:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
val is_zero : int -> bool = <fun>
```

En effet si le l'argument d'appel est différent de `0` la fonction ne sait pas quelle valeur retourner. On peut alors compléter le filtrage en utilisant le motif universel.

```
# let is_zero n = match n with
  0 -> true
  | _ -> false ;;
val is_zero : int -> bool = <fun>
```

Si, à l'exécution, aucun motif n'est sélectionné, alors une exception est déclenchée. Ainsi, on peut écrire :

```
# let f x = match x with 1 -> 3 ;;
Characters 11-30:
Warning: this pattern-matching is not exhaustive.
```

```
Here is an example of a value that is not matched:  
0  
val f : int -> int = <fun>  
# f 1 ;;  
- : int = 3  
# f 4 ;;  
Uncaught exception: Match_failure("", 11, 30)
```

L'exception `Match_Failure` est déclenchée à l'appel de `f 4`, et si elle n'est pas récupérée entraîne l'arrêt du calcul en cours (voir ??)

Combinaison de motifs

La combinaison de plusieurs motifs permet d'obtenir un nouveau motif qui pourra déstructurer une valeur selon l'un ou l'autre de ses motifs originaux. La forme syntaxique est la suivante :

Syntaxe

`p1 | ... | pn` Elle construit un nouveau motif par combinaison des motifs `p1`, ..., et `pn`. La seule contrainte forte est de refuser tout nommage à l'intérieur de ces motifs. Donc chacun d'eux ne devra contenir que des valeurs constantes ou le motif universel. L'exemple suivant montre comment vérifier qu'un caractère est une voyelle.

```
# let est_une_voyelle c = match c with  
  'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true  
  | _ -> false ;;  
val est_une_voyelle : char -> bool = <fun>  
# est_une_voyelle 'i' ;;  
- : bool = true  
# est_une_voyelle 'j' ;;  
- : bool = false
```

Filtrage d'un paramètre

Le filtrage de motif est essentiellement utilisé pour la définition de fonctions par cas. Pour alléger l'écriture de ces définitions, la construction syntaxique **function** autorise le filtrage d'un paramètre :

Syntaxe

function		$p1 \rightarrow \text{expr}_1$
		$p2 \rightarrow \text{expr}_2$
		:
		$p_n \rightarrow \text{expr}_n$

La barre verticale précédant le premier motif est là aussi optionnelle. En fait, tel Mr Jourdain, à chaque fois que nous définissons une fonction, nous utilisons un filtrage de motif. En effet, la construction *function* $x \rightarrow \text{expression}$, est une définition par filtrage utilisant un unique motif réduit à une variable. On peut utiliser cette particularité avec des motifs simples comme dans :

```
# let f = function (x,y) -> 2*x + 3*y + 4 ;;  
val f : int * int -> int = <fun>
```

De fait la forme

function $p1 \rightarrow \text{expr}_1 \mid \dots \mid p_n \rightarrow \text{expr}_n$

est équivalente à

function $\text{expr} \rightarrow \text{match } \text{expr} \text{ with } p1 \rightarrow \text{expr}_1 \mid \dots \mid p_n \rightarrow \text{expr}_n$

En utilisant l'équivalence des déclarations mentionnée page ??, on écrira :

```
# let f (x,y) = 2*x + 3*y + 4 ;;  
val f : int * int -> int = <fun>
```

Mais cette écriture naturelle n'est possible que si la valeur filtrée appartient à un type n'ayant qu'un seul constructeur. Si tel n'est pas le cas, le filtrage n'est pas exhaustif :

```
# let is_zero 0 = true ;;
Characters 13-21:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
val is_zero : int -> bool = <fun>
```

Nommage d'une valeur filtrée

Lors d'un filtrage de motif, il est parfois pratique de nommer tout ou partie du motif. La forme syntaxique suivante introduit le mot clé **as** qui associe un nom à un motif.

Syntaxe

(*pasnom*) Ceci est utile lorsqu'on a besoin de déstructurer une valeur tout en la conservant dans son intégralité. Dans l'exemple suivant, la fonction `min_rat` rend le plus petit rationnel d'un couple de rationnels. Ces derniers sont représentés par un couple numérateur et dénominateur.

```
# let min_rat cr = match cr with
  | (_,0),c2) -> c2
  | (c1,(_,0)) -> c1
  | ((n1,d1) as r1), ((n2,d2) as r2)) ->
    if (n1 * d2) < (n2 * d1) then r1 else r2;;
val min_rat : (int * int) * (int * int) -> int * int = <
```

Pour comparer deux rationnels, il est nécessaire de les déstructurer pour pouvoir nommer leur numérateur et leur dénominateur (`n1`, `n2`, `d1` et `d2`), mais il faut rendre le couple initial (`r1` ou `r2`). La construction **as** nous permet ce nommage de parties d'une même valeur. Cela évite de devoir reconstruire le rationnel retourné en résultat.

Filtrage avec gardes

Le filtrage avec gardes correspond à l'évaluation d'une expression conditionnelle immédiatement après le filtrage d'un motif. Si cette expression renvoie *true*, alors l'expression associée au motif est évaluée, sinon le filtrage continue avec le motif suivant.

Syntaxe

```
match expr with  
:  
| pi when condi -> expri  
:
```

L'exemple suivant utilise deux gardes pour tester l'égalité entre deux rationnels.

```
# let eq_rat cr = match cr with  
  ((_,0), (_,0)) -> true  
  | ((_,0), _) -> false  
  | (_, (_,0)) -> false  
  | ((n1,1), (n2,1)) when n1 = n2 -> true  
  | ((n1,d1), (n2,d2)) when ((n1 * d2) = (n2 * d1)) -> true  
  | _ -> false;;  
val eq_rat : (int * int) * (int * int) -> bool = <fun>
```

Dans le filtrage du quatrième motif, si la garde échoue, le filtrage se poursuit sur le cinquième motif.

Remarque

La vérification de l'exhaustivité du filtrage effectuée par Objective CAML suppose que l'expression conditionnelle de la garde puisse être fausse. En conséquence, elle ne tient pas compte de ce motif puisqu'il n'est pas possible de savoir, avant l'exécution, si la garde sera réalisée ou non.

L'exhaustivité du filtrage dans l'exemple suivant ne pourra pas être détectée.

```
# let f = function x when x = x -> true;;
Characters 10-40:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
_
val f : 'a -> bool = <fun>
```

Filtrage d'intervalles de caractères

Dans le cadre du filtrage sur des caractères, il est fastidieux de construire la combinaison de tous les motifs correspondant à un intervalle de caractères. En effet, si l'on désire tester si un caractère est bien une lettre, il faudra au minimum écrire 26 motifs simples et les combiner. Objective CAML autorise pour les caractères, l'écriture de motifs de la forme :

Syntaxe

'c1' .. 'cn' Elle est équivalente à la combinaison : 'c1' | 'c2' | ... | 'cn'.

Par exemple le motif '0' .. '9' correspond au motif '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'. La première forme est plus agréable à lire et plus rapide à écrire.

Warning

Ce trait fait partie des extensions du langage et pourra évoluer dans les prochaines versions.

En utilisant les motifs combinés et les intervalles, on définit une fonction discriminant les caractères selon plusieurs critères.

```
# let char_discriminate c = match c with
| 'a' | 'e' | 'i' | 'o' | 'u' | 'y'
| 'A' | 'E' | 'I' | 'O' | 'U' | 'Y' -> "Voyelle"
| 'a'..'z' | 'A'..'Z' -> "Consonne"
| '0'..'9' -> "Chiffre"
| _ -> "Autre" ;;
```

```
val char_discriminate : char -> string = <fun>
```

On peut noter que l'ordre des ensembles de motifs a son importance. En effet le second ensemble de motifs contient le premier, mais il n'est examiné qu'après l'échec du premier.

Filtrage des listes

Comme nous l'avons vu, une liste peut être :

- soit vide (la liste est de la forme `[]`),
- soit composée d'un premier élément (sa tête) et d'une sous-liste (sa queue). La liste est alors de la forme `t : q`.

Ces deux écritures possibles d'une liste sont utilisables comme des motifs et permettent de filtrer une liste.

```
# let rec size x = match x with
  [] -> 0
  | _::queue_x -> 1 + (size queue_x) ;;
val size : 'a list -> int = <fun>
# size [];
- : int = 0
# size [7;9;2;6];;
- : int = 4
```

On peut donc reprendre les exemples décrits précédemment (voir page [??](#)) en utilisant le filtrage de motifs, comme par exemple pour l'itération sur les listes.

```
# let rec fold_left f a = function
  [] -> a
  | tete::queue -> fold_left f (f a tete) queue ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
# fold_left (+) 0 [8;4;10];;
- : int = 22
```

Déclaration de valeurs par filtrage d'un motif

La déclaration de valeurs utilise en fait le filtrage de motifs. La déclaration `let x = 18` filtre la valeur 18 avec le motif `x`. Tout motif est accepté comme filtre d'une déclaration, les variables du motif sont associées aux valeurs qu'elles filtrent.

```
# let (a,b,c) = (1, true, 'A');;
val a : int = 1
val b : bool = true
val c : char = 'A'
# let (d,c) = 8, 3 in d + c;;
- : int = 11
```

La portée des variables de filtrage est celle usuelle de la déclaration de valeurs y compris pour les déclarations locales. Ici, `c` reste associée à la valeur `'A'`.

```
# a + (int_of_char c);;
- : int = 66
```

Comme tout filtrage, celui-ci peut ne pas être exhaustif.

```
# let [x;y;z] = [1;2;3];;
Characters 5-12:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val x : int = 1
val y : int = 2
val z : int = 3
# let [x;y;z] = [1;2;3;4];;
Characters 4-11:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
Uncaught exception: Match_failure("", 4, 11)
```

Tout motif est accepté, y compris les constructeurs, le motif universel et les motifs combinés.

```
# let tete :: 2 :: _ = [1; 2; 3] ;;
```

Characters 5-19:

Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:

```
[]  
val tete : int = 1  
# let _ = 3. +. 0.14 in "PI" ;;  
- : string = "PI"
```

Ce dernier exemple a peu d'intérêt dans le monde fonctionnel dans la mesure où la valeur 3.14 calculée n'est pas nommée et donc perdue.

Déclaration de types

Les déclarations de type sont un autre constituant possible d'une phrase Objective CAML. Elles permettent de définir de nouveaux types correspondant aux structures de données originales utilisées dans un programme. Il y a deux grandes familles de types : les types produit pour les n-uplets ou les enregistrements ; les types somme pour les unions.

Les déclarations de type utilisent le mot clé **type**.

Syntaxe

typenom = typedef;; À la différence des déclarations de variables, les déclarations de types sont par défaut récursives. C'est-à-dire que les déclarations de types, quand elles sont combinées, permettent de déclarer des types mutuellement récursifs.

Syntaxe

type	<i>nom1</i>	=	<i>typedef1</i>
and	<i>nom2</i>	=	<i>typedef2</i>
	:		
and	<i>nomn</i>	=	<i>typedefn</i> ;;

Les déclarations de types peuvent être paramétrées par des

variables de type. Un nom de variable de type commence toujours par une apostrophe (caractère ') :

Syntaxe

type'*anom* = *typedef*;; Lorsqu'ils sont plusieurs, les paramètres de types sont déclarés comme un n-uplet devant le nom du type :

Syntaxe

type('a1 ...'an)*nom* = *typedef*;; Seuls les paramètres de type définis dans la partie gauche de la déclaration de type peuvent apparaître dans sa partie droite.

Remarque

L'afficheur de types d'Objective CAML renomme les paramètres de type rencontrés, le premier se nomme '*a*', le deuxième '*b*' et ainsi de suite.

On peut toujours définir un nouveau type à partir d'un type ou de plusieurs types existants.

Syntaxe

typenom = *expression de type* Ceci est utile pour contraindre un type que l'on trouve trop général.

```
# type 'param couple_entier = int * 'param ;;  
type 'a couple_entier = int * 'a  
# type couple_precis = float couple_entier ;;  
type couple_precis = float couple_entier
```

Néanmoins sans contrainte sur les types, l'inférence produira le type le plus général.

```
# let x = (3, 3.14) ;;
```



```
val x : int * float = 3, 3.14
```

Mais on peut utiliser la contrainte de type pour voir apparaître le nom désiré :

```
# let (x:couple_precis) = (3, 3.14) ;;  
val x : couple_precis = 3, 3.14
```

Enregistrements

Les enregistrements sont des n-uplets dont chaque champ est nommé à la manière des *record* de Pascal ou des *struct* de C. Un enregistrement correspond toujours à la déclaration d'un nouveau type. Un type enregistrement est défini par la déclaration de son nom, du nom de chacun de ses champs et de leur type.

Syntaxe

typenom = {nom1:t1; ...;nomn:tn};; On peut définir un type représentant les nombres complexes par :

```
# type complex = { re:float; im:float } ;;  
type complex = { re: float; im: float }
```

La création d'une valeur de type enregistrement s'effectue en donnant une valeur à chacun des champs (dans un ordre quelconque).

Syntaxe

{nomi1 = expr_{i1}; ...;nomin = expr_{in}};; Par exemple, on crée un nombre complexe de partie réelle 2. et de partie imaginaire 3. :

```
# let c = {re=2.;im=3.} ;;  
val c : complex = {re=2; im=3}  
# c = {im=3.;re=2.} ;;
```

```
- : bool = true
```

Dans le cas où des champs sont manquants, l'erreur suivante se produit :

```
# let d = { im=4. } ;;  
Characters 9-18:  
Some labels are undefined
```

L'accès à un champ est possible de deux façons : par la notation pointée ou par le filtrage de certains champs.

La syntaxe de la notation pointée est usuelle :

Syntaxe

expr.nom L'expression *expr* doit être d'un type enregistrement possédant le champ *nom*.

Le filtrage d'un enregistrement permet de connaître la valeur associée à un certain nombre de champs. Un motif de filtrage pour un enregistrement a la syntaxe suivante :

Syntaxe

$\{nom_i=pi; \dots; nom_j=pj\}$ Les motifs sont à droite du symbole $=$ ($pi, \dots, pj$). Il n'est pas nécessaire de faire figurer tous les champs d'un enregistrement dans un tel motif.

La fonction `add_complex` accède aux champs par la notation pointée alors que la fonction `mult_complex` y accède par filtrage.

```
# let add_complex c1 c2 = {re=c1.re+c2.re; im=c1.im+c2.im}
val add_complex : complex -> complex -> complex = <fun>
# add_complex c c ;;
- : complex = {re=4; im=6}
# let mult_complex c1 c2 = match (c1,c2) with
  ({re=x1;im=y1},{re=x2;im=y2}) -> {re=x1*.x2-.y1*.y2;im=
val mult_complex : complex -> complex -> complex = <fun>
```

```
# mult_complex c c ;;
- : complex = {re=-5; im=12}
```

L'intérêt des enregistrements, par rapport aux n-uplets, est au moins double :

- une information descriptive et discriminante grâce aux noms des champs : cela permet en particulier de simplifier les motifs de filtrage;
- un accès identique, par son nom, de n'importe quel champ de l'enregistrement : l'ordre des champs n'a plus d'importance, seul leur nom compte.

L'exemple suivant montre la facilité d'accès aux champs des enregistrements par rapport au n-uplets :

```
# let a = (1,2,3) ;;
val a : int * int * int = 1, 2, 3
# let f tr = match tr with x,_,_ -> x ;;
val f : 'a * 'b * 'c -> 'a = <fun>
# f a ;;
- : int = 1
# type triplet = {x1:int; x2:int; x3:int} ;;
type triplet = { x1: int; x2: int; x3: int }
# let b = {x1=1; x2=2; x3=3} ;;
val b : triplet = {x1=1; x2=2; x3=3}
# let g tr = tr.x1 ;;
val g : triplet -> int = <fun>
# g b ;;
- : int = 1
```

Pour le filtrage de motif, il n'est pas nécessaire d'indiquer tous les champs de l'enregistrement à filtrer. Le type inféré est alors celui du dernier type enregistrement déclaré contenant ces champs.

```
# let h tr = match tr with {x1=x} -> x;;
val h : triplet -> int = <fun>
# h b;;
- : int = 1
```

Il existe une construction permettant de créer un enregistrement

identique à un autre sauf pour quelques champs. C'est souvent utile pour les enregistrements contenant de nombreux champs.

Syntaxe

```
{ nomwithnomi = expri; ...; nomj = exprj }
```

```
# let c = {b with x1=0} ; ;
```

```
val c : triplet = {x1=0; x2=2; x3=3}
```

Une nouvelle copie de la valeur de b est créée où seul le champ x1 a une nouvelle valeur.

Warning

Ce trait fait partie des extensions du langage et pourra évoluer dans les prochaines versions.

Types somme

À la différence des n-uplets ou des enregistrements, qui correspondent à un produit cartésien, la déclaration d'un type somme correspond à une union d'ensembles. On regroupe dans un même type des types différents (par exemple des entiers et des chaînes de caractères). Les différents membres de la somme sont discriminés par des constructeurs qui permettent d'une part, comme leur nom l'indique, de *construire* les valeurs de ce type et d'autre part d'*accéder* aux composantes de ces valeurs grâce au filtrage de motif. Appliquer un constructeur à un argument, c'est indiquer que la valeur retournée appartient à ce nouveau type.

On déclare un type somme en donnant le nom de ses constructeurs et le type de leur éventuel argument.

Syntaxe

type

```
nom = ...  
| Nomi ...  
| Nomj of tj ...  
| Nomk of tk * ... * tl ...;;
```

Un nom de constructeur est un identificateur particulier : Un nom de constructeur est un identificateur particulier :

Warning

Les noms des constructeurs commencent toujours par une majuscule.

Constructeurs constants

On appelle constructeur constant un constructeur qui n'attend pas d'argument. Les constructeurs constants peuvent être par la suite utilisés directement comme valeur du langage, comme une constante.

```
# type piece = Pile | Face;;  
type piece = | Pile | Face  
# Pile;;  
- : piece = Pile
```

Le type *bool* peut se définir de cette manière.

Constructeurs avec arguments

Les constructeurs peuvent avoir des arguments. Le mot clé **of** indique le type des arguments du constructeur. Cela permet de regrouper sous un même type des objets de types différents, chacun étant introduit avec un constructeur particulier.

Voici un exemple classique de définition d'un type de données pour représenter les cartes dans un jeu, ici le Tarot. On définit les types *couleur* et *carte* de la manière suivante :

```
# type couleur = Pique | Coeur | Carreau | Trefle ;;
# type carte =
  Roi of couleur
  | Dame of couleur
  | Cavalier of couleur
  | Valet of couleur
  | Petite_carte of couleur * int
  | Atout of int
  | Excuse ;;
```

La création d'une valeur du type *carte* s'effectue par application d'un constructeur à une valeur de son type.

```
# Roi Pique ;;
- : carte = Roi Pique
# Petite_carte(Coeur, 10) ;;
- : carte = Petite_carte (Coeur, 10)
# Atout 21 ;;
- : carte = Atout 21
```

Et voici, par exemple, la fonction *toutes_les_cartes* qui construit la liste de toutes les cartes d'une couleur passée en argument.

```
# let rec interval a b = if a = b then [b] else a::(interval
val interval : int -> int -> int list = <fun>
# let toutes_les_cartes s =
  let les_figures = [ Valet s; Cavalier s; Dame s; Roi s
  and les_autres = List.map (function n -> Petite_carte (s, n))
  in les_figures @ les_autres ;;
val toutes_les_cartes : couleur -> carte list = <fun>
# toutes_les_cartes Coeur ;;
- : carte list =
[Valet Coeur; Cavalier Coeur; Dame Coeur; Roi Coeur; Petite_carte
  Petite_carte (Coeur, 2); Petite_carte (Coeur, 3); Petite_carte (Coeur, 4);
  ...]
```

Pour manipuler les valeurs des types somme, on utilise le filtrage de motif. L'exemple suivant construit les fonctions de conversion de valeurs de type *couleur* et de type *carte* en chaînes de caractères (type *string*) :

```
# let string_of_couleur = function
| Pique    -> "pique"
| Carreau  -> "carreau"
| Coeur    -> "coeur"
| Trefle   -> "trèfle" ;;
val string_of_couleur : couleur -> string = <fun>
# let string_of_carte = function
| Roi c      -> "roi de " ^ (string_of_couleur c)
| Dame c     -> "dame de " ^ (string_of_couleur c)
| Valet c    -> "valet de " ^ (string_of_couleur c)
| Cavalier c -> "cavalier de " ^ (string_of_couleur c)
| Petite_carte (c, n) -> (string_of_int n) ^ " de " ^ (string_of_couleur c)
| Atout n    -> (string_of_int n) ^ " d'atout"
| Excuse     -> "excuse" ;;
val string_of_carte : carte -> string = <fun>
```

L'alignement des motifs permet une lecture aisée de ces fonctions.

Le constructeur `Petite_carte` est considéré comme un constructeur à *deux* arguments. Le filtrage d'une telle valeur nécessite de nommer ses deux composants.

```
# let est_petite_carte c = match c with
| Petite_carte v -> true
| _ -> false;;
```

Characters 45-59:

The constructor `Petite_carte` expects 2 argument(s), but is here applied to 1 argument(s)

Pour éviter de devoir nommer chaque composant d'un constructeur, on le déclare en ayant un seul argument en parenthésant le type *n*-uplet associé. Le filtrage des deux constructeurs suivants diffère.

```
# type t =
| C of int * bool
| D of (int * bool) ;;
# let acces v = match v with
| C (i, b) -> i, b
| D x -> x;;
val acces : t -> int * bool = <fun>
```

Types récurifs

La définition de types récurifs est indispensable à tout langage algorithmique pour décrire les structures de données usuelles (listes, piles, arbres, graphes, etc.). Pour cela, la définition de types (**type**) est en Objective CAML par défaut réursive, à la différence de la déclaration de valeurs (**let**).

Le type des listes prédéfini en Objective CAML n'admet qu'un seul paramètre. On peut vouloir, dans une structure de liste, stocker des valeurs appartenant à deux types différents, par exemple, des entiers (*int*) ou des caractères (*char*). Dans ce cas, on définit :

```
# type int_or_char_list =  
  Nil  
  |   Int_cons of int * int_or_char_list  
  |   Char_cons of char * int_or_char_list ;;  
  
# let l1 = Char_cons ( '=', Int_cons(5, Nil) ) in  
  Int_cons ( 2, Char_cons ( '+', Int_cons(3, l1) ) ) ;;  
- : int_or_char_list =  
Int_cons (2, Char_cons ('+', Int_cons (3, Char_cons ('='
```

Types paramétrés

Un utilisateur peut également déclarer des types avec paramètres. Ceci permet de généraliser l'exemple des listes contenant des valeurs de deux types différents.

```
# type ('a, 'b) list2 =  
  Nil  
  |   Acons of 'a * ('a, 'b) list2  
  |   Bcons of 'b * ('a, 'b) list2 ;;
```



```
# Acons(2, Bcons('+', Acons(3, Bcons('=', Acons(5, Nil)))
- : (int, char) list2 =
Acons (2, Bcons ('+', Acons (3, Bcons ('=', Acons (...))
```

On peut, évidemment, instancier les paramètres '*a*' et '*b*' avec un même type.

```
# Acons(1, Bcons(2, Acons(3, Bcons(4, Nil)))) ;;
- : (int, int) list2 = Acons (1, Bcons (2, Acons (3, Bco
```

Cette utilisation du type *list2* peut, comme dans l'exemple précédent, servir à marquer les entiers pairs et les entiers impairs. On extrait alors la sous-liste des entiers pairs pour construire une liste ordinaire.

```
# let rec extract_odd = function
  Nil -> []
  | Acons(_, x) -> extract_odd x
  | Bcons(n, x) -> n::(extract_odd x) ;;
val extract_odd : ('a, 'b) list2 -> 'b list = <fun>
```

La définition de cette fonction ne donne aucune indication sur la nature des valeurs stockées dans la structure. C'est pourquoi son type est paramétré.

Portée des déclarations

Les noms de constructeurs obéissent à la même discipline de portée que les déclarations globales : une redéfinition masque l'ancienne. Néanmoins les valeurs d'un type masqué existent toujours. La boucle d'interaction ne distinguera pas ces deux types à l'affichage. D'où certains messages d'erreur peu clairs.

Dans ce premier exemple, le constructeur constant Nil du type *int_or_char* a été masqué par la déclaration des constructeurs du type ('*a*', '*b*') *list2*.

```
# Int_cons(0, Nil) ;;
Characters 13-16:
```

This expression has type ('a, 'b) list2 but is here used with type int_or_char_list

Ce deuxième exemple provoque un message d'erreur assez déroutant, du moins la première fois qu'il apparaît. Soit le petit programme suivant :

```
# type t1 = Vide | Plein;;
type t1 = | Vide | Plein
# let vide_t1 x = match x with Vide -> true | Plein -> false
val vide_t1 : t1 -> bool = <fun>
# vide_t1 Vide;;
- : bool = true
```

Ensuite, nous redéclarons le type *t1* :

```
# type t1 = {u : int; v : int} ;;
type t1 = { u: int; v: int }
# let y = { u=2; v=3 } ;;
val y : t1 = {u=2; v=3}
```

Alors si l'on applique la fonction *vide_t1* sur une valeur du nouveau type *t1*, on obtient le message d'erreur suivant :

```
# vide_t1 y;;
Characters 9-10:
This expression has type t1 but is here used with type t
```

La première occurrence de *t1* représente le premier type défini alors que la deuxième correspond au deuxième type.

Types fonctionnels

Le type de l'argument d'un constructeur peut être quelconque. En particulier, il peut tout aussi bien contenir le type d'une fonction. Le type suivant construit des listes dont tous les éléments sauf le dernier sont des valeurs fonctionnelles.

```
# type 'a listf =
```

```

Val of 'a
| Fun of ('a -> 'a) * 'a listf ;;
type 'a listf = | Val of 'a | Fun of ('a -> 'a) * 'a listf

```

Puisque les valeurs fonctionnelles sont des valeurs manipulables dans le langage, on peut construire des valeurs de type *listf* :

```

# let huit_div = (/) 8 ;;
val huit_div : int -> int = <fun>
# let gl = Fun (succ, (Fun (huit_div, Val 4))) ;;
val gl : int listf = Fun (<fun>, Fun (<fun>, Val 4))

```

et des fonctions filtrant de telles valeurs :

```

# let rec compute = function
  Val v -> v
  | Fun(f, x) -> f (compute x) ;;
val compute : 'a listf -> 'a = <fun>
# compute gl;;
- : int = 3

```

Exemple : représentation des arbres

Les arbres sont des structures récurrentes en programmation. Les types récursifs permettent de définir et manipuler facilement de telles structures. Nous donnons dans ce paragraphe deux exemples de structures arborescentes.

Arbres binaires

On définit une structure d'arbre binaire dont les noeuds sont étiquetés par des valeurs d'un même type en déclarant :

```

# type 'a arbre_bin =
  Empty

```

```
| Node of 'a arbre_bin * 'a * 'a arbre_bin ;;
```

On utilise cette structure pour définir un petit programme de tri utilisant des arbres binaires de recherche. Un arbre binaire de recherche a cette propriété que toutes les valeurs du fils gauche sont inférieures à celle de la racine, et toutes celles du fils droit lui sont supérieures. La figure 2.5 donne un exemple d'une telle structure pour des entiers. Les noeuds vides (constructeur Empty) y sont représentés par des petits carrés ; les autres (constructeur Node), par un cercle où est inscrit la valeur stockée.

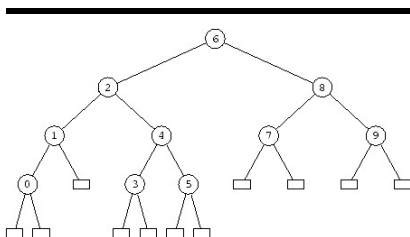


Figure 2.5 : Arbre binaire de recherche

On en extrait d'un arbre binaire de recherche une liste triée par un parcours infixe effectuée par la fonction suivante :

```
# let rec list_of_arbre = function
  Empty -> []
  | Node(fg, r, fd) -> (list_of_arbre fg) @ (r :: (list_of_arbre fd))
val list_of_arbre : 'a arbre_bin -> 'a list = <fun>
```

Pour obtenir à partir d'une liste un arbre binaire de recherche, on définit une fonction d'ajout.

```
# let rec ajout x = function
  Empty -> Node(Empty, x, Empty)
  | Node(fg, r, fd) -> if x < r then Node(ajout x fg, r, fd)
  else Node(fg, r, ajout x fd) ;;
val ajout : 'a -> 'a arbre_bin -> 'a arbre_bin = <fun>
```

La fonction de transformation de liste en arbre s'obtient par itération sur la liste de la fonction ajout.

```
# let rec arbre_of_list = function
  [] -> Empty
  | t::q -> ajout t (arbre_of_list q) ;;
val arbre_of_list : 'a list -> 'a arbre_bin = <fun>
```

La fonction de tri est alors simplement la composition des fonctions `arbre_of_list` et `list_of_arbre`.

```
# let tri x = list_of_arbre (arbre_of_list x) ;;
val tri : 'a list -> 'a list = <fun>
# tri [5; 8; 2; 7; 1; 0; 3; 6; 9; 4] ;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Arbres planaires généraux

Dans cette partie, nous utilisons les fonctions prédéfinies du module `List` (voir page ??) suivantes :

- `List.map` : qui applique une fonction à tous les éléments d'une liste et retourne la liste des résultats;
- `List.fold_left` : qui est une version équivalente de la fonction `fold_left` définie page ??;
- `List.exists` qui applique une fonction à valeur booléenne à tous les éléments d'une liste, si une de ces applications vaut `true` alors le résultat est `true`, sinon la fonction retourne `false`.

Un arbre planaire général est un arbre dont on ne fixe pas a priori le nombre de fils : à chaque noeud on associe une liste de fils dont la longueur peut varier.

```
# type 'a arbre = Empty
  | Node of 'a * 'a arbre list ;;
```

L'arbre vide est représenté par la valeur *Empty*, une feuille est un noeud sans fils soit de la forme *Node(x, [])*, soit de la forme dégénérée *Node(x, [Empty; Empty; ..])*. Il est alors relativement aisé

d'écrire des fonctions de manipulation de tels arbres comme l'appartenance d'un élément à un arbre ou le calcul de la hauteur de l'arbre.

Pour tester l'appartenance d'un élément e , on utilise l'algorithme suivant : si l'arbre est vide alors e n'appartient pas à cet arbre, sinon e appartient à l'arbre si et seulement s'il est égal à l'étiquette de la racine, ou s'il appartient à un des fils.

```
# let rec appartient e = function
  Empty -> false
  | Node(v, fs) -> (e=v) or (List.exists (appartient e) f)
val appartient : 'a -> 'a arbre -> bool = <fun>
```

Pour calculer la hauteur d'un arbre, on utilise la définition suivante : un arbre vide est de hauteur 0 et sinon la hauteur de l'arbre est égale à la hauteur du plus haut sous arbre augmentée de 1 .

```
# let rec hauteur =
  let max_list l = List.fold_left max 0 l in
  function
    Empty -> 0
    | Node (_, fs) -> 1 + (max_list (List.map hauteur fs))
val hauteur : 'a arbre -> int = <fun>
```

Valeurs récursives non fonctionnelles

La déclaration récursive de valeurs non fonctionnelles permet de construire des structures de données circulaires.

La déclaration suivante construit une liste circulaire à un élément.

```
# let rec l = 1::l ;;  
val l : int list =  
[1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1]
```

L'application d'une fonction récursive sur une telle liste risque de

boucler jusqu'à saturation de la mémoire.

```
# size 1 ;;  
Stack overflow during evaluation (looping recursion?).
```

L'égalité structurelle reste utilisable avec de telles listes uniquement lorsque l'égalité physique est d'abord vérifiée :

```
# l=l ;;  
- : bool = true
```

En revanche, si l'on définit une nouvelle liste, pourtant égale, il ne faut pas utiliser le test d'égalité structurelle sous peine de voir votre programme boucler indéfiniment. Nous ne recommandons donc pas de tenter d'évaluer l'exemple suivant :

```
let rec l2 = 1::l2 in l=l2 ;;
```

Par contre, l'égalité physique reste toujours possible.

```
# let rec l2 = 1::l2 in l==l2 ;;  
- : bool = false
```

Le prédicat `==` teste l'égalité d'une valeur immédiate ou le partage d'un objet structuré (égalité sur l'adresse de la valeur). Nous allons nous en servir pour vérifier qu'en explorant une liste nous ne repassons pas par une sous-liste déjà examinée. En premier lieu, nous définissons la fonction `memq` qui vérifie la présence d'un élément dans une liste en s'appuyant sur l'égalité physique. C'est le pendant de la fonction `mem` qui teste elle l'égalité structurelle; ces deux fonctions appartiennent au module `List`.

```
# let rec memq a l = match l with  
  [] -> false  
  | b::l -> (a==b) or (memq a l) ;;  
val memq : 'a -> 'a list -> bool = <fun>
```

La fonction de calcul de la taille est redéfinie en conservant la liste des listes déjà examinées et en s'arrêtant si une liste est rencontrée une seconde fois.

```
# let special_size l =
```

```

let rec size_aux previous l = match l with
| [] -> 0
| _::l1 -> if memq l previous then 0
else 1 + (size_aux (l::previous) l1)
in size_aux [] l ;;
val special_size : 'a list -> int = <fun>
# special_size [1;2;3;4] ;;
- : int = 4
# special_size l ;;
- : int = 1
# let rec l1 = 1::2::l2 and l2 = 1::2::l1 in special_size l1
- : int = 4

```

Typage, domaine de définition et exceptions

Le type inféré d'une fonction correspond à un sur-ensemble de son domaine de définition. Ce n'est pas parce qu'une fonction prend un paramètre de type *int* qu'elle saura calculer une valeur pour tous les entiers passés en argument. On traite en général ce problème en utilisant le mécanisme d'exceptions d'Objective CAML. Le déclenchement d'une exception provoque une rupture du calcul qui peut être interceptée et traitée par le programme. Pour cela l'exécution du programme doit avoir posé un récupérateur d'exception avant le calcul de l'expression qui provoque le déclenchement de cette exception.

Fonctions partielles et exceptions

Le domaine de définition d'une fonction correspond à l'ensemble des valeurs sur lesquelles la fonction effectue son calcul. Il existe de nombreuses fonctions mathématiques partielles, on peut citer la division ou le logarithme népérien. Ce problème se pose aussi pour les fonctions manipulant des structures de données plus complexes. En effet quel est le résultat du calcul du premier élément d'une liste vide ? De la même manière le calcul de la fonction factorielle sur un entier négatif peut entraîner un calcul infini.

Un certain nombre de situations exceptionnelles peuvent se produire durant l'exécution d'un programme, par exemple une

tentative de division par zéro. Tenter de diviser un nombre par zéro provoquera au mieux l'arrêt du programme, au pire un état incohérent de la machine. La *sûreté* d'un langage de programmation passe par la garantie de ne pas se retrouver dans une telle situation pour ces cas particuliers. Les exceptions sont une manière d'y répondre.

La division de 1 par 0 provoquera le déclenchement d'une exception spécifique :

```
# 1/0;;  
Uncaught exception: Division_by_zero
```

Le message `Uncaught exception: Division_by_zero` indique d'une part que l'exception `Division_by_zero` a été déclenchée, et que d'autre part elle n'a pas été récupérée. Cette exception fait partie des déclarations de base du langage.

Souvent, le type d'une fonction ne correspond pas à son domaine de définition quand un filtrage de motif n'est pas exhaustif, c'est à dire qu'il ne filtre pas tous les cas de l'expression donnée. Pour prévenir une telle erreur, Objective CAML affiche un message dans un tel cas.

```
# let tete l = match l with t::q -> t ;;  
Characters 14-36:  
Warning: this pattern-matching is not exhaustive.  
Here is an example of a value that is not matched:  
[]  
val tete : 'a list -> 'a = <fun>
```

Si néanmoins le programmeur maintient sa définition incomplète, Objective CAML utilisera le mécanisme d'exceptions en cas d'appel erroné à la fonction partielle :

```
# tete [] ;;  
Uncaught exception: Match_failure("", 14, 36)
```

Enfin, nous avons déjà rencontré une autre exception prédéfinie : `Failure`. Elle prend un argument de type *string*. On peut déclencher

cette exception en utilisant la fonction `failwith`. On pourra ainsi l'utiliser pour compléter notre définition de la fonction `tete` :

```
# let tete = function
  [] -> failwith "Liste vide"
  | h::t -> h;;
val tete : 'a list -> 'a = <fun>
# tete [] ;;
Uncaught exception: Failure("Liste vide")
```

Définition d'une exception

En Objective CAML, les exceptions appartiennent à un type prédéfini *exn*. Ce type est très particulier puisque c'est un type somme *extensible* : on peut étendre l'ensemble des valeurs du type en déclarant de nouveaux constructeurs. Cette particularité permet à l'utilisateur de définir ses propres exceptions en ajoutant au type *exn* des nouveaux constructeurs.

La syntaxe de la déclaration d'une exception est la suivante :

Syntaxe

`exceptionNom;;` ou

Syntaxe

`exceptionNomof t;` Voici des exemples de déclarations d'exceptions :

```
# exception A_MOI;;
exception A_MOI
# A_MOI;;
- : exn = A_MOI
# exception Depth of int;;
exception Depth of int
```

```
# Depth 4;;  
- : exn = Depth(4)
```

Une exception est donc une valeur du langage à part entière.

Warning

Les noms d'exceptions sont des constructeurs. Ils commencent donc obligatoirement par une majuscule.

```
# exception minuscule ;;  
Characters 11-20:  
Syntax error
```

Warning

Les exceptions sont monomorphes : elles n'ont pas de paramètre de type dans la déclaration du type de leur argument.

```
# exception Value of 'a ;;  
Characters 20-22:  
Unbound type parameter 'a
```

Une exception polymorphe autoriserait la définition de fonctions avec un type de retour quelconque comme nous le verrons plus loin, [page ??](#).

Déclenchement d'une exception

La fonction `raise` est une fonction primitive du langage. Elle prend une exception comme argument et possède un type de retour entièrement polymorphe.

```
# raise ;;  
- : exn -> 'a = <fun>  
# raise A_MOI;;
```

```
Uncaught exception: A_MOI
# 1+(raise A_MOI);;
Uncaught exception: A_MOI
# raise (Depth 4);;
Uncaught exception: Depth(4)
```

Il n'est pas possible d'écrire en Objective CAML la fonction `raise`. Elle doit être prédéfinie.

Récupération d'une exception

Tout l'intérêt de déclencher des exceptions réside dans la capacité de les récupérer et d'orienter la suite du calcul selon la valeur de l'exception déclenchée. L'ordre de calcul d'une expression prend alors de l'importance pour déterminer quelle exception est déclenchée. On sort du cadre purement fonctionnel, où l'ordre d'évaluation des arguments peut changer le résultat d'un calcul comme il sera discuté au chapitre suivant (voir page ??).

La construction syntaxique suivante, qui calcule la valeur d'une expression, permet la récupération d'une exception déclenchée lors de ce calcul :

Syntaxe

```
try expr with
| p1 -> expr1
:
| pn -> exprn
```

Si le calcul de *expr* ne déclenche pas d'exception, alors le résultat est celui du calcul de *expr*. Sinon, la valeur de l'exception déclenchée est filtrée; la valeur de l'expression correspondante à la première branche du filtrage correct est retournée. Si aucune branche du filtrage ne correspond à la valeur de l'exception alors celle-ci se propage jusqu'au précédent **try-with** posé durant l'exécution du programme. Donc le filtrage d'une exception est toujours considéré comme exhaustif. Implicitement, le dernier filtre est `| e -> raise e`.

Si aucun récupérateur d'exception n'est rencontré dans le programme, le système lui-même se charge d'intercepter l'exception et termine le programme en affichant un message d'erreur.

Il ne faut pas confondre *calculer* une exception (c'est-à-dire une valeur de type *exn*) et *déclencher* une exception qui effectue une rupture de calcul. Une exception étant une valeur comme les autres elle peut être rendue comme résultat d'une fonction.

```
# let rendre x = Failure x ;;
val rendre : string -> exn = <fun>
# rendre "test" ;;
- : exn = Failure("test")
# let declencher x = raise (Failure x) ;;
val declencher : string -> 'a = <fun>
# declencher "test" ;;
Uncaught exception: Failure("test")
```

On remarque que l'application de `declencher` ne rend pas de valeur alors que celle de `rendre` en rend une de type *exn*.

Calculer avec des exceptions

Outre leur utilisation pour le traitement de valeurs non désirées, les exceptions permettent aussi un style de programmation et peuvent être source d'optimisations. L'exemple suivant effectue le produit de tous les éléments d'une liste d'entiers. On utilise une exception pour interrompre le parcours de la liste et retourner la valeur 0 dès qu'on la rencontre.

```
# exception Found_zero ;;
exception Found_zero
# let rec mult_rec l = match l with
| [] -> 1
| 0 :: _ -> raise Found_zero
| n :: x -> n * (mult_rec x) ;;
val mult_rec : int list -> int = <fun>
# let mult_list l =
  try mult_rec l with Found_zero -> 0 ;;
```

```
val mult_list : int list -> int = <fun>  
# mult_list [1;2;3;0;5;6] ;;  
- : int = 0
```

Ainsi tous les calculs restant en attente, à savoir les multiplications par n qui suivent chacun des appels récurifs, sont abandonnées. Après avoir rencontré le **raise**, le calcul reprend à partir du filtrage du **with**.

Polymorphisme et valeurs de retour des fonctions

Le polymorphisme paramétrique d'Objective CAML permet de définir des fonctions dont le type de retour est complètement indéterminé. Par exemple :

```
# let id x = x ;;  
val id : 'a -> 'a = <fun>
```

Cependant, le type de retour dépend du type de l'argument. Ainsi, lorsque l'on applique la fonction `id` à un argument, le mécanisme d'inférence de types sait instancier la variable de type `'a`. Donc, pour chaque utilisation particulière, le type de `id` peut être déterminé.

S'il n'en était pas ainsi, l'utilisation du typage statique fort, chargé d'assurer la sûreté d'exécution, n'aurait plus de sens. En effet, une fonction d'un type entièrement indéterminé comme `'a -> 'b` autoriserait n'importe quelle conversion de type qui amènerait inévitablement à une erreur durant l'exécution puisque la représentation physique des valeurs de type différents ne sont pas les mêmes.

Apparente contradiction

Cependant, il est possible de définir dans le langage Objective CAML des fonctions dont le type de retour contient une variable de type n'apparaissant pas dans le type de ses arguments. Nous allons en considérer quelques exemples et voir pourquoi une telle

possibilité n'est pas contradictoire avec le typage statique fort.

Voici un premier exemple :

```
# let f x = [] ;;  
val f : 'a -> 'b list = <fun>
```

Cette fonction permet de construire une valeur polymorphe à partir de n'importe quoi :

```
# f () ;;  
- : '_a list = []  
# f "n'importe quoi" ;;  
- : '_a list = []
```

Néanmoins, la valeur obtenue n'est pas entièrement indéterminée : il s'agit d'une liste. Elle ne pourra donc pas être utilisée n'importe où.

Voici trois exemples de fonctions dont le type est celui redouté 'a -> 'b :

```
# let rec f1 x = f1 x ;;  
val f1 : 'a -> 'b = <fun>  
# let f2 x = failwith "n'importe quoi" ;;  
val f2 : 'a -> 'b = <fun>  
# let f3 x = List.hd [] ;;  
val f3 : 'a -> 'b = <fun>
```

Ces fonctions ne sont, en fait pas dangereuses, vis-à-vis de la sûreté d'exécution car il n'est pas possible de les utiliser pour construire une valeur : la première boucle, les deux dernières déclenchent une exception qui interrompt le calcul.

C'est également pour ne pas pouvoir définir de fonction de type 'a -> 'b que les nouveaux constructeurs d'exceptions ne peuvent pas avoir d'argument dont le type contient une variable.

En effet, si l'on pouvait déclarer une exception polymorphe Poly_exn de type 'a -> exn, on pourrait alors écrire la fonction :

```
let f = function
```

```
0 -> raise (Poly_exn false)
| n -> n+1 ;;
```

La fonction f étant de type $int \rightarrow int$ et $Poly_exn$ étant de type $'a \rightarrow exn$, on pourrait alors définir :

```
let g n = try f n with Poly_exn x -> x+1 ;;
```

Cette fonction est également bien typée (puisque l'argument de $Poly_exn$ peut être quelconque) et, dès lors, l'évaluation de $(g\ 0)$ aboutirait à la tentative d'additionner un entier et un booléen !

Calculatrice de bureau

Pour comprendre comment se bâtit un programme en Objective CAML, il est nécessaire d'en réaliser un. L'exemple choisi est une calculatrice de bureau, le modèle le plus simple, c'est-à-dire qui ne fonctionne que sur des nombres entiers et ne réalise que les quatre opérations standard de l'arithmétique.

Pour commencer, on définit le type *touche* pour représenter les touches d'une calculette. Celle-ci possède quinze touches, à savoir : une pour chaque opération, une pour chaque chiffre et la touche $=$.

```
# type touche = Plus | Moins | Fois | Par | Egal | Chiffre
```

On remarque que les touches numériques sont regroupés sous un unique constructeur *Chiffre* prenant un entier comme argument. De ce fait, certaines valeurs de type *touche* ne représentent pas réellement une touche. Par exemple, $(Chiffre\ 32)$ est une valeur possible du type *touche*, mais ne représente aucune touche de la calculatrice.

On écrit donc une fonction valide qui vérifie que son argument correspond à une touche de la calculatrice. Le type de cette fonction est $touche \rightarrow bool$, c'est-à-dire qu'elle prend comme argument une valeur de type *touche* et rend une valeur de type *bool*.

La première étape est de définir une fonction qui vérifie qu'un

entier est compris entre 0 et 9. Nous déclarons cette fonction sous le nom `est_chiffre` :

```
# let est_chiffre = function x -> (x>=0) && (x<=9) ;;  
val est_chiffre : int -> bool = <fun>
```

On définit ensuite la fonction valide par filtrage sur son argument de type *touche* :

```
# let valide tch = match tch with  
  Chiffre n -> est_chiffre n  
  | _ -> true ;;  
val valide : touche -> bool = <fun>
```

Le premier filtre s'applique quand l'argument de valide est une valeur construite avec le constructeur *Chiffre*; dans ce cas, l'argument de *Chiffre* est testé par la fonction `est_chiffre`. Le second filtre s'applique dans tous les autres cas de valeur du type *touche*. On rappelle que grâce au typage, la valeur filtrée est obligatoirement de type *touche*.

Avant d'entamer le codage du mécanisme de la calculatrice, nous allons déterminer un modèle permettant de décrire d'un point de vue formel la réaction à l'activation d'une des touches de la machine. On considère qu'une calculette possède quatre mémoires où sont respectivement conservés le dernier calcul effectué, la dernière touche actionnée, le dernier opérateur actionné et le nombre affiché à l'écran. L'ensemble de ces quatre mémoires est appelé l'état de la calculatrice, il est modifié par chaque appui sur une touche du clavier. Cette modification s'appelle une transition et la théorie gouvernant ce genre de mécanismes est celle des automates. Un état sera représenté dans notre programme par un type enregistrement :

```
# type etat = {  
  dce : int;      (* dernier calcul effectué      *)  
  dta : touche;  (* dernière touche actionnée    *)  
  doa : touche;  (* dernier opérateur actionné   *)  
  vaf : int      (* valeur affichée              *)  
} ;;
```

La figure 2.6 donne un exemple d'une suite de transitions.

	état	touche
	(0, =, =, 0)	3
$\frac{3}{4}\textcircled{\text{R}}$	(0, 3, =, 3)	+
$\frac{3}{4}\textcircled{\text{R}}$	(3, +, +, 3)	2
$\frac{3}{4}\textcircled{\text{R}}$	(3, 2, +, 2)	1
$\frac{3}{4}\textcircled{\text{R}}$	(3, 1, +, 21)	$\times$
$\frac{3}{4}\textcircled{\text{R}}$	(24, *, *, 24)	2
$\frac{3}{4}\textcircled{\text{R}}$	(24, 2, *, 2)	=
$\frac{3}{4}\textcircled{\text{R}}$	(48, =, =, 48)	

Figure 2.6 : transitions pour $3 + 21 * 2 =$

On a besoin par la suite d'une fonction *evalue* qui prend deux entiers et une valeur de type *touche* contenant un opérateur et qui renvoie le résultat de l'opération correspondant à la touche appliquée aux deux entiers. Cette fonction est définie par filtrage sur son dernier argument, de type *touche* :

```
# let evaluer x y tch = match tch with
  Plus   -> x + y
  | Moins -> x - y
  | Fois   -> x * y
  | Par    -> x / y
  | Egal   -> y
  | Chiffre _ -> failwith "evaluer : no op";;
val evaluer : int -> int -> touche -> int = <fun>
```

On donne maintenant la définition de la fonction de transition en énumérant tous les cas possibles. On suppose que l'état courant est le quadruplet (a, b, op, d) :

- une touche avec le chiffre x est enfoncée, alors deux cas sont à envisager :
 - la dernière touche enfoncée était aussi un chiffre. C'est donc un nombre que l'utilisateur de la calculatrice est en

train de composer, il faut en conséquence ajouter le chiffre x à la valeur affichée, c'est à dire le remplacer par $d \times 10 + x$. Le nouvel état est :
 $(a, (Chiffre\ x), \dot{A}, d \times 10 + x)$

- la dernière touche enfoncée n'était pas un chiffre. C'est donc le début d'un nouveau nombre qui est composé. Le nouvel état est :
 $(a, (Chiffre\ x), \dot{A}, x)$

- une touche avec l'opérateur $\ddot{A}$ a été enfoncée, le deuxième opérande d'une opération a donc été entièrement composé et il s'agit pour la calculatrice d'effectuer cette opération. C'est à cette fin qu'est conservée la dernière opération (ici $\dot{A}$). Le nouvel état est :
 $(a \dot{A} d, \ddot{A}, \ddot{A}, a \dot{A} d)$

Pour écrire la fonction transition, il suffit de traduire mot à mot en Objective CAML la définition précédente : la définition par cas devient une définition par filtrage sur la touche passée en argument. Le cas d'une touche qui comporte lui même deux cas, est traité par la fonction locale `transition_chiffre` par filtrage sur la dernière touche actionnée.

```
# let transition et tou =
  let transition_chiffre n = function
    Chiffre _ -> { et with dta=tou; vaf=et.vaf*10+n }
    | _ -> { et with dta=tou; vaf=n }
  in
  match tou with
    Chiffre p -> transition_chiffre p et.dta
    | _ -> let res = evaluate et.dce et.vaf et.doa
  in { dce=res; dta=tou; doa=tou; vaf=res } ;;
val transition : etat -> touche -> etat = <fun>
```

Cette fonction prend un *etat*, une *touche* et calcule le nouvel *etat*.

On peut maintenant tester ce programme avec l'exemple précédent :

```
# let etat_initial = { dce=0; dta=Egal; doa=Egal; vaf=0
```

```

val etat_initial : etat = {dce=0; dta=Egal; doa=Egal; vaf=0}
# let etat2 = transition etat_initial (Chiffre 3) ;;
val etat2 : etat = {dce=0; dta=Chiffre 3; doa=Egal; vaf=3}
# let etat3 = transition etat2 Plus ;;
val etat3 : etat = {dce=3; dta=Plus; doa=Plus; vaf=3}
# let etat4 = transition etat3 (Chiffre 2) ;;
val etat4 : etat = {dce=3; dta=Chiffre 2; doa=Plus; vaf=6}
# let etat5 = transition etat4 (Chiffre 1) ;;
val etat5 : etat = {dce=3; dta=Chiffre 1; doa=Plus; vaf=9}
# let etat6 = transition etat5 Fois ;;
val etat6 : etat = {dce=24; dta=Fois; doa=Fois; vaf=24}
# let etat7 = transition etat6 (Chiffre 2) ;;
val etat7 : etat = {dce=24; dta=Chiffre 2; doa=Fois; vaf=48}
# let etat8 = transition etat7 Egal ;;
val etat8 : etat = {dce=48; dta=Egal; doa=Egal; vaf=48}

```

Cette exécution peut s'écrire de manière plus concise en utilisant une fonction appliquant une suite de transitions correspondant à une liste de touches passée en argument.

```

# let transition_liste et lt = List.fold_left transition etat_initial lt
val transition_liste : etat -> touche list -> etat = <fun>
# let exemple = [ Chiffre 3; Plus; Chiffre 2; Chiffre 1;
  in transition_liste etat_initial exemple ;;
- : etat = {dce=48; dta=Egal; doa=Egal; vaf=48}

```

Exercices

Fusion de deux listes

1. Écrire une fonction [fusion_i](#) qui prend en entrée deux listes d'entiers triées en ordre croissant et retourne une nouvelle liste triée contenant les éléments des deux premières.

```

# let rec fusion_i l1 l2 = match (l1,l2) with
  [],_ -> l2

```

```

| _, [] -> l1
| t1::q1,t2::q2 ->
if t1 < t2 then t1::(fusion_i q1 l2)
else t2::(fusion_i l1 q2);;
val fusion_i : 'a list -> 'a list -> 'a list = <fun

```

2. Écrire une fonction **fusion** générale, qui prend en argument une fonction de comparaison et deux listes triées selon cet ordre et retourne la liste fusionnée selon le même ordre. La fonction de comparaison sera du type `'a -> 'a -> bool`.

```

# let rec fusion f_ord l1 l2 = match (l1,l2) with
  [],_ -> l2
  | _, [] -> l1
  | t1::q1,t2::q2 ->
    if f_ord t1 t2 then t1::(fusion f_ord q1 l2)
    else t2::(fusion f_ord l1 q2);;
val fusion : ('a -> 'a -> bool) -> 'a list -> 'a li

```

3. Appliquer cette fonction à 2 listes d'entiers triées en ordre décroissant, puis à deux listes de chaînes de caractères triées en ordre décroissant.

```

# fusion (>) [44;33;22;11] [55;30;10];;
- : int list = [55; 44; 33; 30; 22; 11; 10]
# fusion (<) ["mon"; "premier"; "programme"]
  ["bon"; "essai"; "numero"; "un"];;
- : string list =
["bon"; "essai"; "mon"; "numero"; "premier"; "progr

```

4. Que se **passait-il** si une des listes ne respecte pas l'ordre donné?
5. Écrire un nouveau type **liste** sous la forme d'un enregistrement contenant trois champs : la liste classique, une fonction d'ordre et un booléen indiquant si la liste respecte l'ordre.

```
# type 'a liste = {l: 'a list; f_ord : 'a -> 'a -> }
type 'a liste = { l: 'a list; f_ord: 'a -> 'a -> bo
```

6. Écrire la fonction **ajoute** qui ajoute un élément à une liste de ce type.
7. Écrire une fonction **tri** qui trie par insertion les éléments d'une liste.

```
# let ajoute e lt =
  let rec ajoute_rec e l = match l with
  | [] -> [e]
  | t::q -> if lt.f_ord e t then e::l
  else t::(ajoute_rec e q)
  in
  if lt.r then {lt with l=ajoute_rec e lt.l}
  else {lt with l = e::lt.l};;
val ajoute : 'a -> 'a liste -> 'a liste = <fun>
```

```
# let tri lt =
  if lt.r then lt
  else List.fold_right ajoute lt.l {l=[]; f_ord=lt.f
val tri : 'a liste -> 'a liste = <fun>
```

8. Écrire une nouvelle fonction **fusion** pour ces listes.

```
# let rec fusion_lt l1 l2 =
  if l1.f_ord <> l2.f_ord then failwith "ordre inco
  else
  if l1.r then
  if l2.r then {l = fusion l1.f_ord l1.l l2.l; f_ord
  else List.fold_right ajoute l2.l l1
  else
  if l2.r then fusion_lt l2 l1
  else fusion_lt (tri l1) l2;;
```

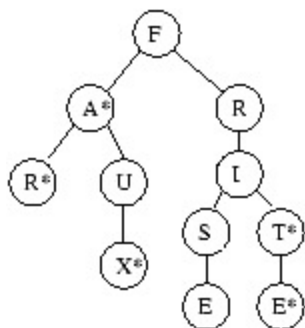
```
val fusion_lt : 'a liste -> 'a liste -> 'a liste =
```

Arbres lexicaux

Pour la représentation de dictionnaires, on utilisera des arbres lexicaux (ou *tries*).

```
# type noeud_lex = Lettre of char * bool * arbre_lex
  and arbre_lex = noeud_lex list;;
# type mot = string;;
```

La valeur booléenne du *noeud_lex* marque la fin d'un mot lorsqu'elle vaut **true**. Dans une telle structure, la suite de mots << fa, far, faux, frise, frit, frite >> est stockée de la façon suivante :



L'étoile (*) marque la fin d'un mot.

1. Écrire la fonction **existe** qui teste si un mot appartient à un dictionnaire de type *arbre_lex*.

```
# let rec existe m d =
  let aux sm i n = match d with
  | [] -> false
  | (Lettre (c,b,l))::q when c=sm.[i] ->
    if n = 1 then b else existe (String.sub sm (i+1) (
```

```

    | (Lettre (c,b,l))::q -> existe sm q
  in aux m 0 (String.length m) ;;
val existe : string -> arbre_lex -> bool = <fun>

```

2. Écrire une fonction **ajoute** qui prend un mot et un dictionnaire et retourne un nouveau dictionnaire qui contient ce mot en plus. Si le mot existe déjà, il n'est pas nécessaire de l'ajouter.

```

# let rec ajoute m d =
  let aux sm i n =
    if n = 0 then d
    else match d with
    [] -> [Lettre (sm.[i], n = 1, ajoute (String.sub sm
    | (Lettre(c,b,l))::q when c=sm.[i]->
    if n = 1 then (Lettre(c,true,l))::q
    else Lettre(c,b,ajoute (String.sub sm (i+1) (n-1))
    | (Lettre(c,b,l))::q -> (Lettre(c,b,l))::(ajoute s
    in aux m 0 (String.length m) ;;
val ajoute : string -> arbre_lex -> arbre_lex = <fun>

```

3. Écrire une fonction **construit** qui prend une liste de mots et construit le dictionnaire correspondant.

```

# let construit l =
  let rec aux l d = match l with
  [] -> d
  | t::q -> aux q (ajoute t d)
  in aux l [] ;;
val construit : string list -> arbre_lex = <fun>

```

4. Écrire une fonction **verifie** qui prend une liste de mots et un dictionnaire et retourne la liste des mots n'appartenant pas à ce dictionnaire.


```
# let rec filter p = function
  [] -> []
  | t::q -> if p t then t::(filter p q) else filter p q
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
# let verifie l d = filter (function x -> not (exists l1 d1
val verifie : string list -> arbre_lex -> string list
```

5. Écrire une fonction [selecte](#) qui prend un dictionnaire et une longueur et retourne l'ensemble des mots de cette longueur.

```
# let string_of_char c = String.make 1 c ;;
val string_of_char : char -> string = <fun>
# let rec selecte n d = match d with
  [] -> []
  | (Lettre(c,b,l))::q when n=1 ->
    let f (Lettre (c,b,_)) = if b then string_of_char c else ""
    filter (function x -> x <> "!") (List.map f d)
  | (Lettre(c,b,l))::q ->
    let r1 = selecte (n-1) l
    and r2 = selecte n q in
    let pr = List.map (function s -> (string_of_char c ^ s)) (r1 @ r2)
    pr @ r2 ;;
val selecte : int -> arbre_lex -> string list = <fun>
```

Parcours de graphes

On définit un type *'a graphe* représentant les graphes orientés par listes d'adjacence contenant pour chaque sommet la liste de ses successeurs :

```
# type 'a graphe = ( 'a * 'a list) list ;;
```

1. Écrire une fonction [ajoute_som](#) qui ajoute un sommet à un graphe et retourne le nouveau graphe.

```
# let rec ajoute_som s g =
  match g with
  | [] -> [(s, [])]
  | (t, _)::_ when t=s -> failwith "sommet existant"
  | sl::q -> sl::(ajoute_som s q) ;;
val ajoute_som : 'a -> ('a * 'b list) list -> ('a *
# let ajoute_som = (ajoute_som : 'a -> 'a graphe ->
val ajoute_som : 'a -> 'a graphe -> 'a graphe = <fu
```

2. Écrire une fonction [ajoute_arc](#) qui ajoute un arc à un graphe possédant déjà ces deux sommets.

```
# let rec ajoute_arc s1 s2 g =
  match g with
  | [] -> failwith "sommet inconnu"
  | (t, la)::q when t=s1 ->
    if List.mem s2 la then failwith "arc existant"
    else (s1, s2::la)::q
  | sl::q -> sl::(ajoute_arc s1 s2 q) ;;
val ajoute_arc : 'a -> 'b -> ('a * 'b list) list ->
<fun>
# let ajoute_arc = (ajoute_arc : 'a -> 'a -> 'a gra
val ajoute_arc : 'a -> 'a -> 'a graphe -> 'a graphe
```

3. Écrire une fonction [arcs_dir](#) qui retourne tous les sommets accessibles à partir d'un sommet donné.

```
# let rec arcs_dir s g = match g with
  | [] -> []
  | (t, la)::_ when t=s -> la
  | _::q -> arcs_dir s q ;;
val arcs_dir : 'a -> ('a * 'b list) list -> 'b list
# let arcs_dir = (arcs_dir : 'a -> 'a graphe -> 'a
val arcs_dir : 'a -> 'a graphe -> 'a list = <fun>
```

4. Écrire une fonction [arcs_vers](#) qui retourne la liste de tous les

sommets atteignant un sommet donné.

```
# let rec arcs_vers s g = match g with
  [] -> []
  | (t,la)::q -> if List.mem s la then t::(arcs_vers
    else (arcs_vers s q) ;;
val arcs_vers : 'a -> ('b * 'a list) list -> 'b list
# let arcs_vers = (arcs_vers : 'a -> 'a graphe -> '
val arcs_vers : 'a -> 'a graphe -> 'a list = <fun>
```

Résumé

Ce chapitre a montré les principaux traits de la programmation fonctionnelle et du polymorphisme paramétrique qui sont deux traits essentiels du langage Objective CAML. La syntaxe des expressions du noyau fonctionnel du langage ainsi que celle des types ont été décrites permettant la réalisation des premiers programmes. Par ailleurs, il a été souligné la différence profonde entre le type d'une fonction et son domaine de définition.

L'introduction du mécanisme d'exceptions permet de résoudre ce problème et introduit déjà un nouveau style de programmation où l'on précise le déroulement des calculs.

Pour en savoir plus

Le modèle de calcul des langages fonctionnels est le λ -calcul, qui a été inventé par Alonzo Church en 1932. Le but de Church était de définir une notion de calculabilité effective au moyen de la λ -définissabilité. Plus tard, il apparut que la notion ainsi introduite était équivalente aux notions de calculabilité au sens de Turing (machine de Turing) et de Gödel-Herbrand (fonctions récursives). Cette coïncidence incite à penser qu'il existe une notion de calculabilité universelle, indépendante des formalismes particuliers : c'est la thèse de Church. Dans ce calcul, les deux seules constructions sont l'abstraction et l'application. Les structures de données (entiers, booléens, couples, ...) peuvent être codées par des λ -termes.

Les langages fonctionnels, dont le premier représentant a été Lisp, implantent ce modèle et l'étendent principalement par des structures de données plus efficaces. Dans un souci d'efficacité, les premiers langages fonctionnels implantaient des modifications physiques de mémoire, ce qui entre autres forçaient la stratégie d'évaluation en évaluation immédiate ou stricte. Dans cette stratégie, les arguments des fonctions sont évalués avant d'être passés à la fonction. C'est en fait plus tard, pour d'autres langages comme Miranda, Haskell ou LML, que la stratégie d'évaluation retardée (paresseuse ou par nécessité) a été implantée pour les langages fonctionnels purs.

Le typage statique, avec inférence de types, a été promu par la famille ML au début des années 80. La page

Lien: http://www.pps.jussieu.fr/~cousinea/Caml/caml_history.html

présente un historique du langage ML. Son modèle de calcul est alors le λ -calcul typé, sous-ensemble du λ -calcul. Il garantit qu'aucune erreur de typage n'arrivera pendant le déroulement du programme. Néanmoins des programmes $< >$ peuvent être rejetés par le système de types de ML. Ces cas arrivent rarement et ces programmes peuvent toujours être réécrits de manière conforme au système de types.

Les deux langages fonctionnels les plus utilisés sont Lisp et ML, représentants des langages fonctionnels impurs. Pour approfondir l'approche fonctionnelle de la programmation, les livres [AS89] et [CM95] présentent chacun un cours général d'informatique utilisant respectivement les langages Scheme (dialecte Lisp) et Caml-Light.

Programmation impérative

Introduction

À la différence de la programmation fonctionnelle où on calcule une valeur par l'application d'une fonction à ses arguments sans se soucier du déroulement des opérations, la programmation

impérative est plus proche de la représentation machine car elle introduit un état mémoire que le déroulement des actions d'un programme va modifier. On appelle instructions ces actions des programmes et un programme impératif est une suite, ou une séquence, d'instructions. L'état mémoire est susceptible d'être modifié à l'exécution de chaque instruction. On considère les opérations d'entrées-sorties comme des modifications de la mémoire, de la mémoire vidéo ou de fichiers.

Ce style de programmation est directement inspiré de la programmation assembleur. On le retrouve dans les premiers langages évolués généralistes (Fortran, C, Pascal, etc.). En Objective CAML les éléments suivants du langage correspondent à ce modèle :

- les structures de données physiquement modifiables, comme les tableaux ou les enregistrements à champs modifiables;
- les opérations d'entrées-sorties;
- les structures de contrôle de l'exécution comme les boucles et les exceptions.

Certains algorithmes s'écrivent plus facilement dans ce style de programmation. On peut citer comme exemple le produit de deux matrices. Même s'il est effectivement possible de le traduire dans une version purement fonctionnelle, où des listes remplacent les vecteurs, cela n'est ni naturel, ni efficace par rapport à une écriture impérative.

L'intérêt d'intégrer ce modèle dans un langage fonctionnel est de pouvoir écrire certains algorithmes dans ce style de programmation quand ceux-ci s'y prêtent. Les deux principaux désavantages par rapport au style purement fonctionnel sont :

- de compliquer le système de types du langage et de rejeter certains programmes qui sans cela seraient considérés comme corrects;
- de devoir tenir compte de la représentation mémoire et de l'ordre des calculs.

Néanmoins, avec quelques règles de prudence dans l'écriture des programmes, le choix entre plusieurs styles de programmation offre

de plus grandes possibilités d'écriture d'algorithmes, ce qui est l'objectif principal des langages de programmation. En outre, un programme écrit dans un style proche de l'algorithme utilisé sera plus simple donc aura plus de chances d'être correct (ou, tout du moins, plus rapidement mis au point).

Pour ces raisons, le langage Objective CAML possède des types de données dont les valeurs sont physiquement modifiables, des structures de contrôle de l'exécution des programmes et une bibliothèque d'entrées-sorties dans un style impératif.

Plan du chapitre

Ce chapitre continue la présentation des éléments de base du langage Objective CAML entamée dans le chapitre précédent mais en s'intéressant cette fois-ci aux constructions impératives. Il se découpe en cinq sections. La première est la plus importante, elle présente les différentes structures de données physiquement modifiables et décrit leur représentation mémoire. La seconde montre assez brièvement les entrées-sorties de base du langage. La troisième section s'intéresse aux nouvelles structures de contrôle itératives. La quatrième section discute de l'impact des traits impératifs sur le déroulement d'un programme, en particulier sur l'ordre d'évaluation des arguments d'une fonction. La dernière section reprend l'exemple de la calculatrice du chapitre précédent pour en faire une calculatrice à mémoire.

Structures de données physiquement modifiables

Les valeurs des types suivants : vecteurs, chaînes de caractères, enregistrements à champs mutables et références sont des valeurs structurées dont les composants peuvent être physiquement modifiés.

Nous avons vu qu'une variable Objective CAML liée à une valeur garde cette valeur jusqu'à la fin de son existence. On ne peut modifier cette liaison que par une redéfinition et, dans ce cas, il ne s'agit pas à proprement parler de la « même » variable puisque une

nouvelle variable de même nom vient masquer l'ancienne qui n'est plus accessible directement mais qui reste inchangée. Avec les valeurs modifiables, on peut changer la valeur associée à une variable sans avoir à redéclarer cette dernière. On a accès à la valeur d'une variable aussi bien en lecture qu'en écriture.

Vecteurs

Les vecteurs, ou tableaux à une dimension, regroupent un nombre connu d'éléments de même type. On peut écrire directement un vecteur en énumérant ses valeurs encadrées par les symboles `[|` et `|]` et séparées par un point virgule à la manière des listes.

```
# let v = [| 3.14; 6.28; 9.42 |] ;;  
val v : float array = [|3.14; 6.28; 9.42|]
```

La fonction de création `Array.create` prend le nombre d'éléments du vecteur, une valeur initiale et retourne un nouveau vecteur.

```
# let v = Array.create 3 3.14;;  
val v : float array = [|3.14; 3.14; 3.14|]
```

L'accès et la modification d'un élément s'effectuent en précisant l'indice de l'élément désigné :

Syntaxe

expr1.(expr2)

Syntaxe

expr1.(expr2) < -expr3 *expr1* doit être un vecteur (type *array*) dont les valeurs sont du type de *expr3*. L'expression *expr2* doit, bien entendu, être de type *int*. La modification est une expression de type *unit*. Le premier élément d'un vecteur a l'indice 0 et le dernier la longueur du vecteur moins 1. Les parenthèses entourant l'expression de l'indice sont obligatoires.

```
# v.(1) ;;
- : float = 3.14
# v.(0) <- 100.0 ;;
- : unit = ()
# v ;;
- : float array = [|100; 3.14; 3.14|]
```

Si l'indice utilisé pour accéder à un élément d'un tableau est en dehors de l'intervalle des indices de celui-ci, alors une exception sera déclenchée au moment de l'accès.

```
# v.(-1) +. 4.0;;
Uncaught exception: Invalid_argument("Array.get")
```

Cette vérification est effectuée au moment de l'exécution du programme, ce qui peut ralentir l'exécution. Néanmoins elle est indispensable pour éviter de remplir des zones mémoire en dehors de l'espace alloué à un vecteur, ce qui provoquerait des erreurs d'exécution graves.

Les fonctions dédiées à la manipulation des tableaux font partie du module `Array` de la bibliothèque standard. On en donne la description au chapitre 8 (page ??). Nous utiliserons, dans les exemples ci-dessous, les trois fonctions suivantes du module `Array` :

- `create` qui crée un vecteur d'une taille donnée avec une valeur initiale donnée.
- `length` qui donne la longueur d'un vecteur.
- `append` qui concatène deux vecteurs.

Partage de valeurs dans un vecteur

Tous les éléments du vecteur contiennent la valeur passée à leur création. Cela implique un partage de cette valeur si celle-ci est une valeur structurée. Créons, par exemple une matrice comme vecteur de vecteurs à l'aide de la fonction `create` du module `Array`.

```
# let v = Array.create 3 0;;
val v : int array = [|0; 0; 0|]
```



```
# let m = Array.create 3 v;;
val m : int array array = [| [|0; 0; 0|]; [|0; 0; 0|]; [|
```

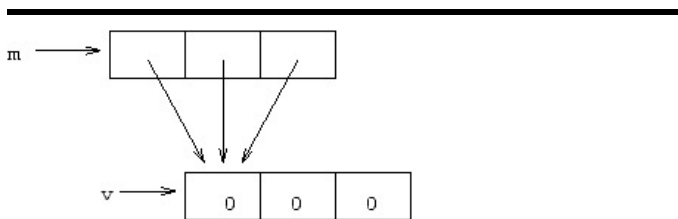


Figure 3.1 : Représentation mémoire d'un vecteur partageant ses éléments

Si l'on modifie l'un des champs du vecteur v ayant servi à la création de m , on modifie du coup toutes les $<>$ de la matrice (voir les figures 3.1 et 3.2).

```
# v.(0) <- 1;;
- : unit = ()
# m;;
- : int array array = [| [|1; 0; 0|]; [|1; 0; 0|]; [|1; 0
```

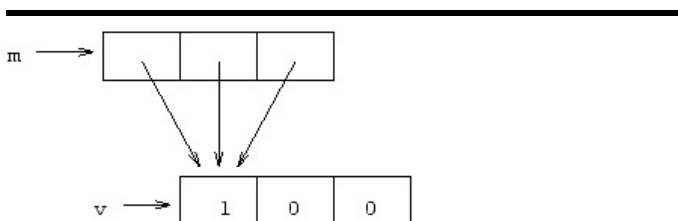


Figure 3.2 : Modification d'éléments d'un vecteur partagé

Il y a duplication si la valeur d'initialisation du vecteur (second argument passé à `Array.create`) est une valeur immédiate et partage

si cette valeur est une valeur structurée. On appelle valeurs immédiates, les valeurs dont la taille ne dépasse pas la taille uniforme des valeurs d'Objective CAML, c'est à dire un mot mémoire. Ce sont les entiers, les caractères, les booléens et les constructeurs constants. Les autres valeurs, dites valeurs structurées, sont représentées par un pointeur sur une zone mémoire. Cette distinction est détaillée au chapitre 9 (page ??).

Les vecteurs de nombres flottants sont un cas particulier. Bien que les nombres flottants soient des valeurs structurées, la création d'un vecteur de flottants effectue une copie de la valeur initiale. Cela pour des questions d'optimisation. Le chapitre 12 sur l'interface avec le langage C (page ??) décrit ce cas particulier.

Matrices non rectangulaires

Une matrice, vecteur de vecteur, peut ne pas être rectangulaire. En effet rien n'empêche de modifier un des vecteurs éléments par un vecteur d'une autre longueur. Cela est utile pour limiter la taille d'une telle matrice. La valeur `t` suivante construit une matrice triangulaire pour les coefficients du triangle de Pascal.

```
# let t = [|
  [|1|];
  [|1; 1|];
  [|1; 2; 1|];
  [|1; 3; 3; 1|];
  [|1; 4; 6; 4; 1|];
  [|1; 5; 10; 10; 5; 1|]
|] ;;
val t : int array array =
  [| [|1|]; [|1; 1|]; [|1; 2; 1|]; [|1; 3; 3; 1|]; [|1; 4; 6; 4; 1|]; [|1; 5; 10; 10; 5; 1|] |]
# t.(3) ;;
- : int array = [|1; 3; 3; 1|]
```

Dans cet exemple, l'élément du vecteur `t` à l'indice `i` est un vecteur d'entiers de taille `i+1`. Pour manipuler de telles matrices, il est nécessaire de calculer la taille de chaque élément vecteur.

Copie de vecteurs

Quand on copie un vecteur ou que l'on concatène deux vecteurs, le résultat obtenu est un nouveau vecteur. Une modification des vecteurs originaux n'entraîne pas de modification des copies, sauf, comme toujours à modifier les valeurs partagées.

```
# let v2 = Array.copy v ;;
val v2 : int array = [|1; 0; 0|]
# let m2 = Array.copy m ;;
val m2 : int array array = [| [|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|] |]
# v.(1) <- 352;;
- : unit = ()
# v2;;
- : int array = [|1; 0; 0|]
# m2 ;;
- : int array array = [| [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|] |]
```

On s'aperçoit dans cet exemple que la copie de m ne copie que les pointeurs vers v. Si un des éléments de v est modifié, alors m2 le sera aussi. La concaténation crée un nouveau vecteur dont la taille est égale à la somme des tailles des deux autres.

```
# let mm = Array.append m m ;;
val mm : int array array =
  [| [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|] |]
  [|1; 352; ...|]; ...|]
# Array.length mm ;;
- : int = 6
# m.(0) <- Array.create 3 0 ;;
- : unit = ()
# m ;;
- : int array array = [| [|0; 0; 0|]; [|1; 352; 0|]; [|1; 352; 0|] |]
# mm ;;
- : int array array =
  [| [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|] |]
  [|1; 352; ...|]; ...|]
```

Par contre la modification de v, valeur partagée dans m et mm, aura un effet sur ces deux matrices.

```
# v.(1) <- 18 ;;
- : unit = ()
# mm;;
- : int array array =
[ [|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; 0|]
...|]
```

Chaînes de caractères

Les chaînes de caractères peuvent être considérées comme un cas spécial de vecteurs de caractères. Néanmoins pour des questions d'occupation mémoire¹ leur type est spécialisé. De plus l'accès aux éléments possède une syntaxe particulière :

Syntaxe

expr1.*[expr2]* Les éléments d'une chaîne de caractères peuvent être modifiés physiquement :

Syntaxe

expr1.*[expr2]* <-*-expr3*

```
# let s = "salut";;
val s : string = "salut"
# s.[2];;
- : char = 'l'
# s.[2]<-'Z';;
- : unit = ()
# s;;
- : string = "saZut"
```

Champs modifiables des enregistrements

Les champs d'un enregistrement peuvent être déclarés modifiables. Il suffit de l'indiquer dans la déclaration du type de l'enregistrement par le mot clé **mutable**.

Syntaxe

typenom = { ...; mutable nomi: t; ... } Voici un petit exemple définissant un type d'enregistrement pour les points du plan :

```
# type point = { mutable xc : float; mutable yc : float  
type point = { mutable xc: float; mutable yc: float }  
# let p = { xc = 1.0; yc = 0.0 } ;;  
val p : point = {xc=1; yc=0}
```

La valeur d'un champ déclaré **mutable** est donc modifiable suivant la syntaxe :

Syntaxe

expr1.nom <- expr2 L'expression *expr1* doit être d'un type enregistrement possédant la champ *nom*. L'opération de modification retourne une valeur de type *unit*.

```
# p.xc <- 3.0 ;;  
- : unit = ()  
# p ;;  
- : point = {xc=3; yc=0}
```

On peut écrire une fonction de déplacement d'un point, modifiant les coordonnées de celui-ci. On utilise la déclaration locale par filtrage pour séquentialiser les effets de bord.

```
# let moveto p dx dy =  
  let () = p.xc <- p.xc +. dx  
  in p.yc <- p.yc +. dy ;;  
val moveto : point -> float -> float -> unit = <fun>  
# moveto p 1.1 2.2 ;;  
- : unit = ()  
# p ;;
```

```
- : point = {xc=4.1; yc=2.2}
```

Il est possible de mélanger des champs modifiables et des champs non modifiables dans la définition d'un enregistrement. Seuls ceux spécifiés **mutable** sont alors modifiables :

```
# type t = { c1 : int; mutable c2 : int } ;;  
type t = { c1: int; mutable c2: int }  
# let r = { c1 = 0; c2 = 0 } ;;  
val r : t = {c1=0; c2=0}  
# r.c1 <- 1 ;;  
Characters 0-9:  
The label c1 is not mutable  
# r.c2 <- 1 ;;  
- : unit = ()  
# r ;;  
- : t = {c1=0; c2=1}
```

Nous donnons page ?? un exemple d'utilisation des enregistrements à champs modifiables et des tableaux pour implanter une structure de pile.

Références

Objective CAML fournit un type polymorphe *ref* qui peut être vu comme le type des pointeurs sur une valeur quelconque ; en Objective CAML, on parlera plutôt de référence sur une valeur. Une valeur référencée peut être modifiée. Le type *ref* est défini comme un enregistrement à un champ modifiable :

```
type 'a ref = {mutable contents:'a}
```

Ce type est muni de raccourcis syntaxiques prédéfinis. On construit une référence sur une valeur par la fonction *ref*. La valeur référencée peut être atteinte par la fonction préfixe (!). La fonction modifiant le contenu d'une référence est la fonction infix (*:=*).

```
# let x = ref 3 ;;  
val x : int ref = {contents=3}  
# x ;;
```

```

- : int ref = {contents=3}
# !x ;;
- : int = 3
# x := 4 ;;
- : unit = ()
# !x ;;
- : int = 4
# x := !x+1 ;;
- : unit = ()
# !x ;;
- : int = 5

```

Polymorphisme et valeurs modifiables

Le type *ref* est paramétré. C'est ce qui permet de l'utiliser pour créer des références sur des valeurs de n'importe quel type. Cependant, il faut apporter quelques restrictions au type des valeurs référencées : on ne peut pas autoriser la création d'une référence sur une valeur de type polymorphe sans prendre quelques précautions.

Supposons qu'il n'y ait aucune restriction, on pourrait alors déclarer :

```
let x = ref [] ;;
```

La variable *x* serait alors de type '*a list ref*' et l'on pourrait en modifier la valeur de façon incohérente avec le typage statique fort d'Objective CAML :

```

x := 1 :: !x ;;
x := true :: !x ;;

```

On aurait ainsi une même variable de type *int list* à un moment et de type *bool list* à un autre.

Pour éviter une telle situation, le mécanisme d'inférence de type d'Objective CAML utilise une nouvelle catégorie de variables de

type : les variables de type faible. Syntaxiquement, elles sont distinguées par le caractère souligné (_) qui les préfixe.

```
# let x = ref [] ;;  
val x : 'a list ref = {contents=[]}
```

La variable de type 'a n'est pas un paramètre de type, mais une inconnue en attente d'instanciation : c'est la première utilisation de x après sa déclaration qui fixera la valeur que prendra 'a et ce, dans tous les types qui en dépendent et de façon définitive :

```
# x := 0::!x ;;  
- : unit = ()  
# x ;;  
- : int list ref = {contents=[0]}
```

La variable x est désormais de type *int list ref*.

Un type contenant une inconnue est en fait monomorphe bien que son type n'ait pas été précisé. Il n'est pas possible d'instancier cette inconnue par un type polymorphe.

```
# let x = ref [] ;;  
val x : 'a list ref = {contents=[]}  
# x := (function y -> ())::!x ;;  
- : unit = ()  
# x ;;  
- : ('a -> unit) list ref = {contents=[<fun>]}
```

Dans cet exemple, bien que nous ayons instancié l'inconnue de type avec un type a priori polymorphe ('a -> *unit*), le type est resté monomorphe avec une nouvelle inconnue de type.

Cette restriction du polymorphisme n'est pas réservée aux références mais s'applique à toute valeur contenant une partie modifiable : les vecteurs et les enregistrements ayant au moins un champ déclaré **mutable**, etc. Tous les paramètres de type, même ceux sans rapport avec une composante modifiable sont alors des

variables de type faible.

```
# type ('a, 'b) t = { ch1 : 'a list ; mutable ch2 : 'b list }  
type ('a, 'b) t = { ch1: 'a list; mutable ch2: 'b list }  
# let x = { ch1 = [] ; ch2 = [] } ;;  
val x : ('_a, '_b) t = {ch1=[]; ch2=[]}
```

Warning

Cette modification du typage de l'application a des conséquences sur les programmes fonctionnels purs.

Lorsque l'on applique une valeur polymorphe à une fonction polymorphe, on obtient également une variable de type faible, car il ne faut pas exclure que la fonction construise des valeurs physiquement modifiables. En d'autres termes, le résultat d'une application est toujours monomorphe.

```
# (function x -> x) [] ;;  
- : '_a list = []
```

On obtient le même résultat avec l'application partielle :

```
# let f a b = a ;;  
val f : 'a -> 'b -> 'a = <fun>  
# let g = f 1 ;;  
val g : '_a -> int = <fun>
```

Pour retrouver un type polymorphe, il faut abstraire le second argument de f puis l'appliquer :

```
# let h x = f 1 x ;;  
val h : 'a -> int = <fun>
```

En effet, l'expression qui définit h est l'expression fonctionnelle **function** x -> f 1 x. Son évaluation produira une fermeture qui ne risque pas de produire d'effet de bord puisque le corps de la fonction n'est pas évalué.

De façon général, on distingue les expressions dites << non expansives >>, dont on est sûr que le calcul ne risque pas

d'effectuer un effet de bord, des autres, dites « expansives ». Le système de types d'Objective CAML classe les expressions du langage selon leur forme syntaxique :

- les expressions « non expansives » incluent principalement les variables, les constructeurs de valeurs non modifiables et l'abstraction;
- les expressions « expansives » incluent principalement l'application, les constructeurs de valeurs modifiables. Il faut y rajouter des structures de contrôles telles la conditionnelle ou le filtrage.

Entrées-sorties

Les fonctions d'entrées-sorties calculent une valeur (souvent de type *unit*), mais durant ce calcul elles effectuent une modification de l'état des périphériques d'entrées-sorties : modification du buffer du clavier, affichage à l'écran, écriture dans un fichier ou modification du pointeur de lecture. Les deux types suivants sont prédéfinis : *in_channel* et *out_channel* pour respectivement les canaux d'entrées et de sorties. Quand une fin de fichier est rencontrée l'exception *End_of_file* est déclenchée. Enfin, les trois constantes suivantes correspondent aux canaux standard d'entrée, de sortie et d'erreur à la manière d'Unix : *stdin*, *stdout* et *stderr*.

Canaux

Les fonctions d'entrées-sorties de la bibliothèque standard d'Objective CAML manipulent des canaux de communication : valeurs de types *in_channel* ou *out_channel*. En dehors des trois valeurs standard prédéfinies, la création d'un canal utilise une des fonctions suivantes :

```
# open_in;;  
- : string -> in_channel = <fun>  
# open_out;;  
- : string -> out_channel = <fun>
```

`open_in` ouvre le fichier s'il existe², déclenche l'exception `Sys_error` sinon. `open_out` crée le fichier indiqué s'il n'existe pas ou l'écrase s'il existe.

```
# let ic = open_in "koala";;  
val ic : in_channel = <abstr>  
# let oc = open_out "koala";;  
val oc : out_channel = <abstr>
```

Les fonctions de fermeture des canaux sont :

```
# close_in ;;  
- : in_channel -> unit = <fun>  
# close_out ;;  
- : out_channel -> unit = <fun>
```

Lecture et écriture

Les fonctions les plus générales de lecture et d'écriture sont les suivantes :

```
# input_line ;;  
- : in_channel -> string = <fun>  
# input ;;  
- : in_channel -> string -> int -> int -> int = <fun>  
# output ;;  
- : out_channel -> string -> int -> int -> unit = <fun>
```

- `input_line ic` : lit sur un canal d'entrée `ic` tous les caractères jusqu'au premier retour chariot ou la fin de fichier, et les rend sous la forme d'une chaîne de caractères (le retour chariot exclu).
- `input ic s p l` : tente de lire `l` caractères sur un canal d'entrée `ic`, et les stocke dans la chaîne `s` à partir du *pième* caractère. Le nombre de caractères effectivement lus est retourné.
- `output oc s p l` : écrit sur un canal de sortie `oc` une partie de la chaîne `s` commençant en *p*-ième caractère de longueur `l`.

Les fonctions suivantes lisent sur l'entrée standard ou écrivent sur la sortie standard :

```
# read_line ;;
- : unit -> string = <fun>
# print_string ;;
- : string -> unit = <fun>
# print_newline ;;
- : unit -> unit = <fun>
```

D'autres valeurs de type simple peuvent être aussi directement lues ou affichées. Ce sont les valeurs des types convertibles en chaînes de caractères.

Déclarations locales et ordre d'évaluation

Nous pouvons simuler une suite d'affichages des expressions de la forme *let x = e1 in e2*. Sachant que, en général, x est une variable locale susceptible d'être utilisée dans e2, nous savons que e1 est évaluée en premier et qu'ensuite vient le tour d'e2. Si les deux expressions sont des fonctions impératives dont le résultat est () mais qu'elles ont eu des effets de bord, nous les avons alors exécutées dans le bon ordre. En particulier puisque nous connaissons la valeur de retour de e1 : la constante () de type unit, on obtient une séquence d'affichage en écrivant la suite de déclarations par filtrage du motif ().

```
# let () = print_string "et un," in
  let () = print_string " et deux," in
  let () = print_string " et trois" in
  print_string " zéro";;
et un, et deux, et trois zéro- : unit = ()
```

Exemple : C+ /C-

L'exemple suivant implante le jeu « C'est plus/C'est moins » qui consiste à cacher un nombre qu'un utilisateur devra trouver. Le

programme indique à chaque réponse si le nombre caché est plus petit ou plus grand que le nombre proposé.

```
# let rec cpcm n =  
  let () = print_string "taper un nombre : " in  
  let i = read_int ()  
  in  
  if i = n then  
    let () = print_string "BRAVO" in  
    let () = print_newline ()  
    in print_newline ()  
  else  
    let () =  
      if i < n then  
        let () = print_string "C+"  
        in print_newline ()  
      else  
        let () = print_string "C-"  
        in print_newline ()  
    in cpcm n ;;  
val cpcm : int -> unit = <fun>
```

Voici un exemple de session :

```
# cpcm 64;;  
taper un nombre : 88  
C-  
taper un nombre : 44  
C+  
taper un nombre : 64  
BRAVO  
  
- : unit = ()
```

Structures de contrôle

Les entrées-sorties et les valeurs modifiables produisent des effets de bord. Leur utilisation est facilitée par un style de programmation impératif muni de nouvelles structures de contrôles. Nous présentons dans ce paragraphe la structure séquentielle et les

structures itératives. Nous avons déjà rencontré la structure de contrôle conditionnelle à la page ?? dont la forme abrégée **ifthen** prend son sens dans le monde impératif. On écrira, par exemple :

```
# let n = ref 1 ;;  
val n : int ref = {contents=1}  
# if !n > 0 then n := n-1 ;;
```

Characters 20-21:

This expression has type int ref but is here used with t

Séquence

La première de des structures typiquement impérative est la séquence. Elle permet l'évaluation ordonnée de gauche à droite d'une suite d'expressions séparées par un point-virgule.

Syntaxe

expr1; ...;exprn Une séquence est une expression dont la valeur est celle de sa dernière expression (ici, *exprn*). Néanmoins toutes les expressions sont évaluées, et en particulier leurs effets de bord sont pris en compte.

```
# print_string "2 = "; 1+1 ;;  
2 = - : int = 2
```

Avec effet de bord, on retrouve la construction usuelle des langages impératifs.

```
# let x = ref 1 ;;  
val x : int ref = {contents=1}  
# x:=!x+1 ; x:=!x*4 ; !x ;;  
- : int = 8
```

Comme les valeurs précédant un point-virgule sont oubliées, Objective CAML émet un avertissement quand elles ne sont pas de type *unit*.

```
# print_int 1; 2 ; 3 ;;
Characters 14-15:
Warning: this expression should have type unit.
1- : int = 3
```

Pour éviter ce message, on peut utiliser la fonction ignore :

```
# print_int 1; ignore 2; 3 ;;
1- : int = 3
```

Un message différent est obtenu si la valeur possède un type fonctionnel car Objective CAML soupçonne que vous avez oublié un paramètre d'une fonction.

```
# let g x y = x := y ;;
val g : 'a ref -> 'a -> unit = <fun>
# let a = ref 10;;
val a : int ref = {contents=10}
# let u = 1 in g a ; g a u ;;
Characters 13-16:
Warning: this function application is partial,
maybe some arguments are missing.
- : unit = ()
# let u = !a in ignore (g a) ; g a u ;;
- : unit = ()
```

En règle générale on parenthèse les séquences pour en clarifier la portée. Syntaxiquement, le parenthésage peut prendre deux formes :

Syntaxe

(expr)

Syntaxe

begin*expr***end** On peut maintenant écrire de façon plus naturelle le programme C+ /C- de la page ?? :

```
# let rec cpcm n =
```

```

print_string "taper un nombre : ";
let i = read_int () in
if i = n then print_string "BRAVO\n\n"
else
begin
if i < n then print_string "C+\n" else print_string "C-
cpcm n
end ;;
val cpcm : int -> unit = <fun>

```

Boucles

Les structures de contrôle itératives sont elles aussi en dehors du monde fonctionnel. La condition de répétition, ou de sortie, d'une boucle n'a de sens que si une modification physique de la mémoire permet d'en changer la valeur. Il existe deux structures de contrôle itératives : la boucle **for** pour une itération bornée et la boucle **while** pour une itération non bornée. Les structures de boucle elles-mêmes sont des expressions du langage. Elles retournent donc une valeur : la constante () du type *unit*. La boucle **for** peut être croissante (**to**) ou décroissante (**downto**) d'un pas de un.

Syntaxe

```

for nom = expr1 to expr2 do expr3 done
for nom = expr1 downto expr2 do expr3 done

```

Les expressions *expr1* et *expr2* sont de type *int*. Si *expr3* n'est pas de type *unit*, le compilateur engendre un message d'avertissement.

```

# for i=1 to 10 do print_int i; print_string " " done; p
1 2 3 4 5 6 7 8 9 10
- : unit = ()
# for i=10 downto 1 do print_int i; print_string " " don
10 9 8 7 6 5 4 3 2 1
- : unit = ()

```


La boucle non bornée est la boucle « tant que » dont la syntaxe est :

Syntaxe

while*expr1***do***expr2***done** L'expression *expr1* doit être de type *bool*. Et, comme pour la boucle **for**, si *expr2* n'est pas de type *unit*, le compilateur engendre un message d'avertissement.

```
# let r = ref 1
  in while !r < 11 do
    print_int !r ;
    print_string " " ;
    r := !r+1
  done ;;
1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

Il est important de comprendre que les boucles sont des expressions comme les autres qui calculent la valeur () du type *unit*.

```
# let f () = print_string "-- fin\n" ;;
val f : unit -> unit = <fun>
# f (for i=1 to 10 do print_int i; print_string " " done
1 2 3 4 5 6 7 8 9 10 -- fin
- : unit = ()
```

Notez que la chaîne "-- fin\n" est affichée après qu'aient été affichés les entiers de 1 à 10 : on a là la manifestation que les arguments (ici la boucle) sont évalués avant d'être passés à la fonction.

En programmation impérative, le corps d'une boucle (l'expression *expr*) ne calcule pas de valeur, mais procède par effets de bords. En Objective CAML, lorsque le corps d'une boucle n'est pas de type *unit*, le compilateur affiche un avertissement, comme pour la séquence :

```
# let s = [5; 4; 3; 2; 1; 0] ;;
val s : int list = [5; 4; 3; 2; 1; 0]
# for i=0 to 5 do List.tl s done ;;
```

Characters 17-26:

Warning: this expression should have type unit.

```
- : unit = ()
```

Exemple : implantation de piles

La structure de données *'a pile* sera implantée sous forme d'un enregistrement contenant un tableau d'éléments et la première position libre dans ce tableau. Voici le type correspondant :

```
# type 'a pile = { mutable ind:int; taille:int; mutable
```

Le champ *taille* contient la taille maximale de la pile.

Les opérations sur ces piles seront *init_pile* pour l'initialisation d'une pile, *empile* pour empiler un élément dans une pile et *depile* pour retourner le sommet de pile et le dépiler.

```
# let init_pile n = {ind=0; taille=n; elts = [| |]} ;;  
val init_pile : int -> 'a pile = <fun>
```

Cette fonction ne peut pas créer un tableau non vide car il faudrait lui fournir la valeur avec lequel le construire. C'est pour cela que le champ *elts* reçoit un tableau vide.

Deux exceptions sont déclarées pour gérer les tentatives de dépilement d'une pile vide et d'ajout d'un élément sur une pile pleine. Elles sont utilisées dans les fonctions *depile* et *empile*.

```
# exception Pile_vide ;;  
# exception Pile_pleine ;;
```

```
# let depile p =  
  if p.ind = 0 then raise Pile_vide  
  else (p.ind <- p.ind - 1; p.elts.(p.ind)) ;;  
val depile : 'a pile -> 'a = <fun>
```

```
# let empile e p =
  if p.elts = [| |] then
    (p.elts <- Array.create p.taille e;
     p.ind <- 1)
  else if p.ind >= p.taille then raise Pile_pleine
  else (p.elts.(p.ind) <- e; p.ind <- p.ind + 1) ;;
val empile : 'a -> 'a pile -> unit = <fun>
```

Voici un petit exemple d'utilisation de cette structure :

```
# let p = init_pile 4 ;;
val p : '_a pile = {ind=0; taille=4; elts=[| |]}
# empile 1 p ;;
- : unit = ()
# for i = 2 to 5 do empile i p done ;;
Uncaught exception: Pile_pleine
# p ;;
- : int pile = {ind=4; taille=4; elts=[|1; 2; 3; 4|]}
# depile p ;;
- : int = 4
# depile p ;;
- : int = 3
```

Si l'on désire éviter que l'ajout d'un élément dans une pile provoque le déclenchement de l'exception `Pile_pleine`, on peut agrandir le tableau. Pour cela le champ `taille` doit être également modifiable :

```
# type 'a pile =
  {mutable ind:int ; mutable taille:int ; mutable elts :
# let init_pile n = {ind=0; taille=max n 1; elts = [| |]}
# let n_empile e p =
  if p.elts = [| |]
  then
  begin
    p.elts <- Array.create p.taille e;
    p.ind <- 1
  end
  else if p.ind >= p.taille then
  begin
    let nt = 2 * p.taille in
    let nv = Array.create nt e in
```

```

for j=0 to p.taille-1 do nv.(j) <- p.elts.(j) done ;
p.elts <- nv;
p.taille <- nt;
p.ind <- p.ind + 1
end
else
begin
p.elts.(p.ind) <- e ;
p.ind <- p.ind + 1
end ;;
val n_empile : 'a -> 'a pile -> unit = <fun>

```

Il faut néanmoins faire attention aux structures de données pouvant s'agrandir à outrance. Voici un petit exemple où la pile initiale grandit selon les besoins.

```

# let p = init_pile 4 ;;
val p : '_a pile = {ind=0; taille=4; elts=[|]|}
# for i = 1 to 5 do n_empile i p done ;;
- : unit = ()
# p ;;
- : int pile = {ind=5; taille=8; elts=[|1; 2; 3; 4; 5; 5|]}
# p.taille ;;
- : int = 8

```

Il est d'usage d'ajouter à la fonction `depile` la possibilité de diminuer la taille de la pile pour éviter d'utiliser trop d'espace mémoire.

Exemple : calcul sur les matrices

Dans cet exemple on cherche à définir un type pour les matrices, tableaux à deux dimensions contenant des nombres flottants, et à écrire quelques opérations sur les matrices. Le type *mat* monomorphe est un enregistrement contenant les dimensions et les éléments de la matrice. Les fonctions `créer_mat`, `accès_mat` et `mod_mat` sont respectivement les fonctions de création, d'accès à un élément et de modification d'un élément.

```

# type mat = { n:int; m:int; t: float array array };;
type mat = { n: int; m: int; t: float array array }

```

```

# let creer_mat n m = { n=n; m=m; t = Array.create_matrix n m 0.0 }
val creer_mat : int -> int -> mat = <fun>
# let acces_mat m i j = m.t.(i).(j) ;;
val acces_mat : mat -> int -> int -> float = <fun>
# let mod_mat m i j e = m.t.(i).(j) <- e ;;
val mod_mat : mat -> int -> int -> float -> unit = <fun>
# let a = creer_mat 3 3 ;;
val a : mat = {n=3; m=3; t=[ [|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|] ]}
# mod_mat a 1 1 2.0; mod_mat a 1 2 1.0; mod_mat a 2 1 1.0;
- : unit = ()
# a ;;
- : mat = {n=3; m=3; t=[ [|0; 0; 0|]; [|0; 2; 1|]; [|0; 1; 1|] ]}

```

La somme de deux matrices a et b est une matrice c telle que $c_{ij} = a_{ij} + b_{ij}$.

```

# let add_mat p q =
  if p.n = q.n && p.m = q.m then
    let r = creer_mat p.n p.m in
    for i = 0 to p.n-1 do
      for j = 0 to p.m-1 do
        mod_mat r i j (p.t.(i).(j) +. q.t.(i).(j))
      done
    done ;
  r
  else failwith "add_mat : dimensions incompatibles";;
val add_mat : mat -> mat -> mat = <fun>
# add_mat a a ;;
- : mat = {n=3; m=3; t=[ [|0; 0; 0|]; [|0; 4; 2|]; [|0; 2; 2|] ]}

```

Le produit de deux matrices a et b est une matrice c telle que $c_{ij} = \sum_{k=1}^m a_{ik} . b_{kj}$

```

# let mul_mat p q =
  if p.m = q.n then
    let r = creer_mat p.n q.m in
    for i = 0 to p.n-1 do
      for j = 0 to q.m-1 do
        let c = ref 0.0 in
        for k = 0 to p.m-1 do
          c := !c +. (p.t.(i).(k) *. q.t.(k).(j))
        done
      done
    done
  r

```

```

done;
mod_mat r i j !c
done
done;
r
else failwith "mul_mat : dimensions incompatibles" ;;
val mul_mat : mat -> mat -> mat = <fun>
# mul_mat a a;;
- : mat = {n=3; m=3; t=[|0; 0; 0|]; [|0; 5; 2|]; [|0;

```

Ordre d'évaluation des arguments

Dans un langage fonctionnel pur, l'ordre d'évaluation des arguments n'a pas d'importance. Comme il n'y a ni modification d'état mémoire, ni rupture de calcul, il n'y a aucun risque d'influence du calcul d'un argument sur un autre. Par contre en Objective CAML, où il existe des valeurs modifiables physiquement et des exceptions, il y a péril à ne pas tenir compte de l'ordre d'évaluation des arguments. L'exemple suivant est spécifique à la version 2.04 d'Objective CAML pour Linux sur machine Intel :

```

# let new_print_string s = print_string s; String.length s
val new_print_string : string -> int = <fun>
# (+) (new_print_string "Hello ") (new_print_string "World")
World!Hello - : int = 12

```

L'affichage des deux chaînes montre que la deuxième chaîne s'affiche avant la première.

Il en est de même avec les exceptions :

```

# try (failwith "fonction") (failwith "argument") with F
- : string = "argument"

```

Si l'on désire préciser l'ordre d'évaluation des arguments, il est nécessaire d'effectuer des déclarations locales forçant cet ordre avant l'appel de la fonction. L'exemple précédent peut alors se réécrire ainsi :

```
# let e1 = (new_print_string "Hello ")
  in let e2 = (new_print_string "World!")
  in (+) e1 e2 ;;
Hello World!- : int = 12
```

En Objective CAML, l'ordre d'évaluation des arguments n'est pas spécifié. En fait, à ce jour, toutes les mises en oeuvre d'Objective CAML évaluent les arguments de droite à gauche. Néanmoins l'utilisation de ce trait d'implantation peut se révéler dangereux pour de futures versions du langage modifiant l'implantation.

On retrouve l'éternel débat sur la conception des langages. Faut-il ne pas spécifier volontairement certains traits du langage et demander aux programmeurs de ne pas s'en servir sous peine de résultats différents de leur programme selon les mises en oeuvre des compilateurs, ou faut-il tout spécifier et donc autoriser les programmeurs à utiliser le langage dans son ensemble quitte à compliquer la tâche de mise en oeuvre et interdire certaines optimisations ?

Calculatrice à mémoire

On reprend maintenant l'exemple de la calculatrice décrite dans le chapitre précédent mais en la dotant cette fois d'une interface utilisateur rendant notre programme propre à être utilisé comme une calculatrice de bureau. Cette boucle permet d'entrer les opérations directement et de voir s'afficher les résultats sans avoir à appliquer explicitement la fonction de transition pour chaque pression d'une touche.

Nous ajoutons quatre nouvelles touches : **C** qui remet à 0 l'affichage, **M** qui met en mémoire un résultat, **m** qui restitue cette mémoire et **OFF** qui « éteint » la calculatrice. Ceci correspond au type suivant :

```
# type touche = Plus | Moins | Fois | Par | Egal | Chiffre
  | MemoireIn | MemoireOut | Clear | Off ;;
```

Il faut alors définir une fonction de traduction des caractères frappés au clavier en des valeurs de type *touche*. L'exception

Touche_non_valide permet de traiter le cas des caractères ne représentant pas une touche de la calculatrice. La fonction code du module Char traduit un caractère en son code ASCII.

```
# exception Touche_non_valide ;;
exception Touche_non_valide
# let traduction c = match c with
  '+' -> Plus
  | '-' -> Moins
  | '*' -> Foix
  | '/' -> Par
  | '=' -> Egal
  | 'C' | 'c' -> Clear
  | 'M' -> MemoireIn
  | 'm' -> MemoireOut
  | 'o' | 'O' -> Off
  | '0'..'9' as c -> Chiffre ((Char.code c) - (Char.code
  | _ -> raise Touche_non_valide ;;
val traduction : char -> touche = <fun>
```

Dans un style impératif, la fonction de transition ne calculera plus un nouvel état, mais modifiera physiquement l'état de la calculatrice. Il faut donc redéfinir le type etat de façon à ce que ses champs soient modifiables. Enfin, on définit l'exception Touche_off pour traiter l'activation de la touche OFF.

```
# type etat = {
  mutable dce : int;      (* dernier calcul effectué
  mutable dta : bool;     (* vrai si touche = chiffre
  mutable doa : touche;   (* dernier opérateur actionné
  mutable vaf : int;      (* valeur affichée
  mutable mem : int       (* mémoire de la calculatrice

# exception Touche_off ;;
exception Touche_off
# let transition et tou = match tou with
  Clear -> et.vaf <- 0
  | Chiffre n -> et.vaf <- ( if et.dta then et.vaf*10+n
  et.dta <- true
  | MemoireIn -> et.dta <- false ;
  et.mem <- et.vaf
```



```

| MemoireOut -> et.dta <- false ;
et.vaf <- et.mem
| Off -> raise Touche_off
| _ -> let dce = match et.doa with
Plus -> et.dce + et.vaf
| Moins -> et.dce - et.vaf
| Fois -> et.dce * et.vaf
| Par -> et.dce / et.vaf
| Egal -> et.vaf
| _ -> failwith "transition: filtre impossible"
in
et.dce <- dce ;
et.dta <- false ;
et.doa <- tou ;
et.vaf <- et.dce;;
val transition : etat -> touche -> unit = <fun>

```

On définit la fonction `go` qui lance la calculette. Sa valeur de retour est `()` puisque ne nous importe que l'effet produit par l'exécution sur l'environnement (entrée/sortie, modification de l'état). Son argument est également la constante `()` puisque la calculette est autonome (elle définit elle-même son état initial) et interactive (les données du calcul sont rentrées au clavier au fur et à mesure des besoins). Les transitions s'effectuent dans une boucle infinie (**whiletrue**do) dont on pourra sortir par l'exception `Touche_off`.

```

# let go () =
  let etat = { dce=0; dta=false; doa=Egal; vaf=0; mem=0 }
  in try
  while true do
  try
  let entree = traduction (input_char stdin)
  in transition etat entree ;
  print_newline () ;
  print_string "affichage : " ;
  print_int etat.vaf ;
  print_newline ()
  with
  Touche_non_valide -> () (* pas d'effet *)
done

```

```
with  
  Touche_off -> () ;;  
val go : unit -> unit = <fun>
```

Notons que l'état initial doit être soit passé en paramètre, soit déclaré localement à l'intérieur de la fonction `go` pour qu'il soit initialisé à chaque application de cette fonction. Si nous avions utilisé une valeur `etat_initial` comme dans le programme fonctionnel, la calculatrice repartirait dans le même état que celui qu'elle avait lorsqu'elle a été coupée. Il aurait été alors difficile d'utiliser deux calculatrices dans le même programme.

Exercices

Listes doublement chaînées

La programmation fonctionnelle se prête bien à la manipulation de structures de données non circulaires, comme par exemple les listes. Par contre pour les structures circulaires, il y a de réelles difficultés d'implantation. On se propose ici de définir des listes doublement chaînées, c'est-à-dire que chaque élément d'une liste connaît son successeur et son prédécesseur.

1. Définir un [type](#) pour des listes doublement chaînées paramétré, en utilisant au moins un enregistrement avec champs mutables.

```
# type 'a cell = {  
  info : 'a ;  
  mutable prev : 'a dlist ;  
  mutable next : 'a dlist }
```

2. Écrire les fonctions [add](#) et [remove](#) qui ajoute et enlève un élément dans une liste doublement chaînée.

```
# let add x = function
```

```

Empty -> Cell { info=x ; prev=Empty ; next=Empty }
| Cell c as l ->
let new_cell = { info=x ; prev=c.prev ; next=l } in
let new_dlist = Cell new_cell in
c.prev <- new_dlist ;
( match new_cell.prev with
Empty -> ()
| Cell pl -> pl.next <- new_dlist ) ;
new_dlist ;;
val add : 'a -> 'a dlist -> 'a dlist = <fun>

# let remove_cell = function
Empty -> failwith "enleve"
| Cell c -> match (c.prev , c.next) with
Empty , Empty -> Empty
| Cell c1 as l , Empty -> c1.next <- Empty ; l
| Empty , ((Cell c2) as l) -> c2.prev <- Empty ; l
| Cell c1 as l1 , (Cell c2 as l2) -> c1.next <- l2
val remove_cell : 'a dlist -> 'a dlist = <fun>

```

Résolution de systèmes linéaires

Cet exercice porte sur le calcul matriciel. Il résoud un système d'équations selon la méthode de Gauss (dite du pivot). On note le système d'équation $AX = Y$ avec A une matrice carrée de dimension n , Y un vecteur de données de dimension n et X un vecteur inconnu de même dimension.

Cette méthode consiste à transformer le système $AX = Y$ en un système équivalent $CX = Z$ tel que la matrice C soit triangulaire supérieure. On diagonalise C pour obtenir la solution.

1. Définir un type *vect*, un type *mat* et un type *syst*.

```

# type vect = float array ;;
type vect = float array

```

```
# type mat = vect array ;;
type mat = vect array
# type syst = { m:mat ; v:vect } ;;
type syst = { m: mat; v: vect }
```

2. Écrire les fonctions utilitaires de manipulation de vecteurs :
[affichage](#) à l'écran d'un système, [addition](#) de deux vecteurs,
[multiplication](#) d'un vecteur par un scalaire.

```
# let my_print_float s =
  let x = string_of_float s
  in let y = match String.length x with
    5 -> x
    | n when n<5 -> (String.make (5-n) ' ') ^ x
    | n -> String.sub x 0 5
  in print_string y ;;
val my_print_float : float -> unit = <fun>

# let add_vect v1 v2 =
  let l = Array.length v1
  in let res = Array.create l 0.0
  in for i=0 to l-1 do res.(i) <- v1.(i) +. v2.(i) done
  res ;;
val add_vect : float array -> float array -> float

# let mult_scal_vect x v =
  let l = Array.length v
  in let res = Array.create l 0.0
  in for i=0 to l-1 do res.(i) <- v.(i) *. x done ;
  res ;;
val mult_scal_vect : float -> float array -> float
```

3. Écrire les fonctions utilitaires de calcul sur les matrices :
[multiplication](#) de deux matrices, [produit](#) d'une matrice par un
 vecteur.

```

# let mult_mat m1 m2 =
  let l1 = Array.length m1 and l2 = Array.length m2.
  and l3 = Array.length m2
  in let res = Array.create_matrix l1 l2 0.0
  in for i=0 to l1-1 do
    for j=0 to l2-1 do
      for k=0 to l3-1 do
        res.(i).(j) <- res.(i).(j) +. m1.(i).(k) *. m2.(k)
      done done done ;
    res ;;
val mult_mat : float array array -> float array array = <fun>

```

```

# let mult_mat_vect m v =
  let l1 = Array.length m and l2 = Array.length v
  in let res = Array.create l1 0.0
  in for i=0 to l1-1 do
    for j=0 to l2-1 do
      res.(i) <- res.(i) +. m.(i).(j) *. v.(j)
    done done ;
  res ;;
val mult_mat_vect : float array array -> float array = <fun>

```

4. Écrire les fonctions utilitaires de manipulation de systèmes :
- [division](#) d'une ligne d'un système par un pivot (A_{ii}),
 - [permutation](#) de deux lignes.

```

# let div_syst s i =
  let p = s.m.(i).(i)
  in s.m.(i).(i) <- 1.0 ;
  for j=i+1 to (Array.length s.m.(0)) - 1 do
    s.m.(i).(j) <- s.m.(i).(j) /. p
  done ;
  s.v.(i) <- s.v.(i) /. p ;;
val div_syst : syst -> int -> unit = <fun>

```

```

# let permut_syst s i j =

```

```

let aux1 = s.m.(i) and aux2 = s.v.(i)
in s.m.(i) <- s.m.(j) ;
s.v.(i) <- s.v.(j) ;
s.m.(j) <- aux1 ;
s.v.(j) <- aux2 ;;
val permut_syst : syst -> int -> int -> unit = <fun>

```

5. Écrire la **diagonalisation** d'un système. En déduire une fonction **résolvant** un système linéaire.

```

# exception Not_linear ;;
exception Not_linear
# let trigonalize s =
  if s.m=[| |] || s.v=[| |] then raise Not_linear ;
  let l = Array.length s.m
  in if l<>Array.length s.m.(0) || l<>Array.length s.v
  for i=0 to l-1 do
    if s.m.(i).(i)=0.0 then
      begin
        let j = ref (i+1)
        in while !j<l && s.m.(!j).(i)=0.0 do incr j done ;
        if !j=l then raise Not_linear ;
        permut_syst s i !j
      end ;
      div_syst s i ;
      for j=i+1 to l-1 do
        s.v.(j) <- s.v.(j) -. s.m.(j).(i) *. s.v.(i) ;
        s.m.(j) <- add_vect s.m.(j) (mult_scal_vect (-. s.v.(i)) s.m.(j))
      done
    done ;;
val trigonalize : syst -> unit = <fun>

```

```

# let solve s =
  trigonalize s ;
  let l = Array.length s.v
  in let res = Array.copy s.v
  in for i = l-1 downto 0 do

```

```

let x = ref res.(i)
in for j=i+1 to l-1 do x := !x -. s.m.(i).(j) *. r
res.(i) <- !x
done ;
res ;;
val solve : syst -> float array = <fun>

```

6. Tester vos fonctions sur les systèmes suivants :

AX $\begin{bmatrix} \square & \square \\ \square & \square \\ \square & \square \\ \square & \square \end{bmatrix} * \begin{bmatrix} \square \\ \square \\ \square \\ \square \end{bmatrix} = \begin{bmatrix} \square \\ \square \\ \square \\ \square \end{bmatrix} = Y$

AX $\begin{bmatrix} \square & \square \\ \square & \square \\ \square & \square \\ \square & \square \end{bmatrix} * \begin{bmatrix} \square \\ \square \\ \square \\ \square \end{bmatrix} = \begin{bmatrix} \square \\ \square \\ \square \\ \square \end{bmatrix} = Y$

AX $\begin{bmatrix} \square & \square \\ \square & \square \\ \square & \square \\ \square & \square \end{bmatrix} * \begin{bmatrix} \square \\ \square \\ \square \\ \square \end{bmatrix} = \begin{bmatrix} \square \\ \square \\ \square \\ \square \end{bmatrix} = Y$

```

# let ax1 = { m = [| [| 10.0 ; 7.0 ; 8.0 ; 7.0 |]
; [| 7.0 ; 5.0 ; 6.0 ; 5.0 |]
; [| 8.0 ; 6.0 ; 10.0 ; 9.0 |]
; [| 7.0 ; 5.0 ; 9.0 ; 10.0 |] |] ;
v = [| 32.0 ; 23.0 ; 33.0 ; 31.0 |] } ;;
val ax1 : syst =
{m=[| [|10; 7; 8; 7|]; [|7; 5; 6; 5|]; [|8; 6; 10;
# let r1 = solve ax1 ;;
val r1 : float array = [|1; 1; 1; 1|]

```

```

# let ax2 = { m = [| [| 10.0 ; 7.0 ; 8.0 ; 7.0 |]
; [| 7.0 ; 5.0 ; 6.0 ; 5.0 |]
; [| 8.0 ; 6.0 ; 10.0 ; 9.0 |]
; [| 7.0 ; 5.0 ; 9.0 ; 10.0 |] |] ;
v = [| 32.1 ; 22.9 ; 33.1 ; 30.9 |] } ;;
val ax2 : syst =
{m=[| [|10; 7; 8; 7|]; [|7; 5; 6; 5|]; [|8; 6; 10;

# let r2 = solve ax2 ;;
val r2 : float array = [|9.2; -12.6; 4.5; -1.1|]

# let ax3 = { m = [| [| 10.0 ; 7.0 ; 8.1 ; 7.
; [| 7.08 ; 5.04 ; 6.0 ; 5.0 |]
; [| 8.0 ; 5.98 ; 9.89 ; 9.0 |]
; [| 6.99 ; 4.99 ; 9.0 ; 9.98 |] |] ;
v = [| 32.0 ; 23.0 ; 33.0 ; 31.0 |] } ;;
val ax3 : syst =
{m=
[| [|10; 7; 8.1; 7.2|]; [|7.08; 5.04; 6; 5|]; [|8;
...|];
v=...}

# let r3 = solve ax3 ;;
val r3 : float array = [| -80.9999999999; 137; -33.9

```

7. Que peut-on [dire](#) des résultats obtenus?

Résumé

Ce chapitre a montré l'intégration des principaux traits de la programmation impérative (valeurs physiquement modifiables, entrées-sorties, structures de contrôle itératives) dans un langage fonctionnel. Seules les valeurs mutables, comme les chaînes, les vecteurs et les enregistrements à champs mutables, peuvent être physiquement modifiées. Les autres valeurs, une fois créées, sont immuables. On obtient ainsi des valeurs en lecture seule (RO) pour

la partie fonctionnelle et des valeurs en lecture-écriture (RW) pour la partie impérative.

Il est à noter que, si l'on ne se sert pas des traits impératifs du langage, cette extension au noyau fonctionnel ne change pas la partie fonctionnelle, en dehors de considérations de typage contournables.

Pour en savoir plus

La programmation impérative est le style de programmation qui a été le plus utilisé depuis les premiers langages informatiques comme Fortran, C ou Pascal. Pour cette raison de nombreux algorithmes sont décrits dans ce style, souvent en utilisant un pseudo-Pascal. Bien qu'ils soient implantables dans le style fonctionnel, l'utilisation de vecteurs incite à utiliser le style impératif. Les structures de données et les algorithmes présentés dans les ouvrages classiques d'algorithmique, comme [AHU85] et [FGS90], sont directement transposables dans le style approprié. Un intérêt supplémentaire de l'inclusion de ces deux styles dans un même langage est de pouvoir définir de nouveaux modèles de programmation par mélange des deux. C'est justement le propos du prochain chapitre.

Styles fonctionnel et impératif

Introduction

Les langages de programmation fonctionnelle et de programmation impérative s'opposent principalement sur le contrôle de l'exécution des programmes et la gestion mémoire des données.

- un programme fonctionnel calcule une expression. Ce calcul donne lieu à la production d'une valeur. L'ordre dans lequel les opérations nécessaires à ce calcul sont effectuées et la représentation physique des données manipulées sont indifférents puisque le résultat est identique dans tous les cas. Dans un tel cadre, la récupération de la mémoire est gérée

implicitement par le langage : on fait appel à un récupérateur automatique de mémoire ou GC¹.

- un programme impératif est une suite d'instructions modifiant un état mémoire. Chaque étape de l'exécution est encadrée par des structures de contrôle rigides indiquant la prochaine instruction à exécuter. Les programmes impératifs manipulent plus souvent des pointeurs ou des références sur des valeurs que les valeurs elles-mêmes. Il faut alors allouer et récupérer explicitement l'espace mémoire nécessaire au stockage des valeurs. Cela entraîne parfois des erreurs d'accès mémoire. Rien n'empêche néanmoins d'utiliser un GC.

Les langages impératifs fournissent un plus grand contrôle de l'exécution et de la représentation mémoire des données. Étant plus proche de la machine réelle, le code peut être plus efficace mais perdre en sûreté d'exécution. La programmation fonctionnelle, parce qu'elle offre un plus haut niveau d'abstraction, procure un meilleur niveau de sûreté d'exécution : le typage (dynamique ou statique) peut y être renforcé, évitant ainsi des opérations sur des valeurs incohérentes ; la récupération automatique de mémoire, quitte à perdre en efficacité, assure l'existence effective des valeurs manipulées.

Historiquement, ces deux paradigmes de programmation se sont vus consignés dans des univers distincts : les applications symboliques pour le premier, et numériques pour le second. Mais certaines choses ont évolué, en particulier les techniques de compilation des langages fonctionnels et l'efficacité des GC. D'autre part, la sûreté d'exécution est devenue un critère important, voire prépondérant, pour la qualité d'une application. En atteste « l'argument de vente » du langage *Java* selon lequel l'efficacité ne doit pas prendre le pas sur la sûreté, tout en restant raisonnablement bonne. Et c'est une idée qui progresse dans le monde des producteurs de logiciel.

Objective CAML se situe dans cette lignée. Il marie les deux paradigmes de programmation permettant ainsi d'accroître son domaine d'application en facilitant l'écriture d'algorithmes dans un style ou dans l'autre. Il conserve néanmoins de bonnes propriétés de sûreté d'exécution grâce à son typage statique, son GC et son

mécanisme d'exception. Les exceptions sont une première structure de contrôle d'exécution explicite; elles permettent d'effectuer une rupture/reprise de calcul. Ce trait est à la frontière de ces deux modèles car bien qu'il ne change pas le résultat d'un calcul, il peut modifier l'ordre d'exécution. Le fait d'introduire des données physiquement modifiables peut changer le comportement de la partie purement fonctionnelle du langage. En effet l'ordre d'évaluation des arguments d'une fonction devient déterminant si cette évaluation effectue des effets de bord. Pour cette raison, de tels langages sont dits < < langages fonctionnels impurs > >. On perd en effet en niveau d'abstraction car le programmeur doit tenir compte du modèle mémoire ainsi que du déroulement du programme. Cela n'est pas toujours négatif, en particulier pour l'efficacité du code écrit. Par contre, les traits impératifs modifient le système de types du langage : certains programmes fonctionnels, correctement typés d'un point de vue théorique, ne le sont pas en pratique du fait de l'introduction des références. Néanmoins, de tels programmes pourront se réécrire facilement.

Plan du chapitre

Ce chapitre effectue une comparaison des modèles fonctionnel et impératif du langage Objective CAML tant au niveau du contrôle d'exécution que de la représentation mémoire des valeurs. Le mélange de ces deux styles permet la construction de nouvelles structures de données. La première section étudie cette comparaison par l'exemple. La deuxième section discute les éléments de choix entre composition de fonctions et séquence d'une part, partage ou copie de valeur d'autre part. La troisième section dégage l'intérêt de mélanger ces deux styles pour la création de données fonctionnelles modifiables permettant ainsi la construction de données non complètement évaluées. La quatrième section décrit les streams, flots de données potentiellement infinis, et leur intégration, *via* le filtrage, dans le langage.

Comparaison entre fonctionnel et impératif

Les chaînes de caractères (*string*) et les listes chaînées (*'a list*) vont nous servir d'exemple pour illustrer les différences entre

« fonctionnel » et « impératif ».

Du côté du fonctionnel

La fonction `map` (voir page ??) est l'une des plus classiques des langages fonctionnels. Dans un style fonctionnel pur, elle s'écrit :

```
# let rec map f l = match l with
| [] -> []
| t::q -> (f t) :: (map f q) ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

La liste résultat de l'application de `f` aux éléments de la liste passée en argument est construite récursivement en spécifiant indépendamment sa tête (`f t`) et sa queue (`map f q`). En particulier, le programme ne précise pas lequel des deux sera calculé en premier.

De plus, la représentation physique des listes n'a pas besoin d'être connue du programmeur pour écrire une telle fonction. En particulier, les problèmes d'allocation et de partage des données sont gérés implicitement par le système et non par le programmeur. L'exemple ci-dessous en est une illustration :

```
# let exemple = [ "un" ; "deux" ; "trois" ] ;;
val exemple : string list = ["un"; "deux"; "trois"]
# let resultat = map (function x -> x) exemple ;;
val resultat : string list = ["un"; "deux"; "trois"]
```

Les listes `exemple` et `resultat` contiennent des valeurs égales :

```
# exemple = resultat ;;
- : bool = true
```

Ces deux valeurs ont exactement la même structure bien que leur représentation en mémoire diffère, comme on s'en rend compte en utilisant le test d'égalité physique :

```
# exemple == resultat ;;
- : bool = false
```

```
# (List.tl exemple) == (List.tl resultat) ;;
- : bool = false
```

Du côté impératif

Reprenons l'exemple précédent, et modifions une chaîne de la liste resultat.

```
# (List.hd resultat).[1] <- 's' ;;
- : unit = ()
# resultat ;;
- : string list = ["us"; "deux"; "trois"]
# exemple ;;
- : string list = ["us"; "deux"; "trois"]
```

Manifestement, cette opération a modifié la liste exemple. Donc, il est indispensable de connaître la structure physique des deux listes manipulées dès que nous utilisons des traits impératifs.

Voyons maintenant comment l'ordre d'évaluation des arguments d'une fonction peut constituer un piège dans un programme impératif. Nous définissons une structure de liste modifiable avec les fonctions de base de création, rajout et accès :

```
# type 'a ilist = { mutable c : 'a list } ;;
type 'a ilist = { mutable c: 'a list }
# let icreate () = { c = [] }
  let iempty l = (l.c = [])
  let icons x y = y.c <- x::y.c ; y
  let ihd x = List.hd x.c
  let itl x = x.c <- List.tl x.c ; x ;;
val icreate : unit -> 'a ilist = <fun>
val iempty : 'a ilist -> bool = <fun>
val icons : 'a -> 'a ilist -> 'a ilist = <fun>
val ihd : 'a ilist -> 'a = <fun>
val itl : 'a ilist -> 'a ilist = <fun>
# let rec imap f l =
```

```

if iempty l then icreate()
else icons (f (ihd l)) (imap f (itl l)) ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>

```

Bien qu'ayant repris l'architecture générale de la fonction map du paragraphe précédent, nous obtenons avec, imap, un résultat sensiblement différent :

```

# let exemple = icons "un" (icons "deux" (icons "trois"
val exemple : string ilist = {c=["un"; "deux"; "trois"]}]
# imap (function x -> x) exemple ;;
Uncaught exception: Failure("hd")

```

Que s'est-il passé ? Simplement, l'évaluation de (itl l) a eu lieu avant celle de (ihd l) de sorte que, à la dernière itération de imap, la liste référencée par l devient la liste vide avant que l'on n'examine sa tête. La liste exemple est désormais définitivement vide bien que nous n'ayons obtenu aucun résultat :

```

# exemple ;;
- : string ilist = {c=[]}

```

Le défaut de la fonction imap provient d'un mélange insuffisamment contrôlé des genres : nous avons laissé au système le choix de l'ordre d'évaluation. Nous pouvons reformuler la fonction imap en explicitant l'ordre d'évaluation à l'aide de la construction syntaxique **let .. in ..**

```

# let rec imap f l =
  if iempty l then icreate()
  else let h = ihd l in icons (f h) (imap f (itl l)) ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
# let exemple = icons "un" (icons "deux" (icons "trois"
val exemple : string ilist = {c=["un"; "deux"; "trois"]}]
# imap (function x -> x) exemple ;;
- : string ilist = {c=["un"; "deux"; "trois"]}

```

Cependant, la liste initiale a ici encore été perdue :

```

# exemple ;;
- : string ilist = {c=[]}

```

Une seconde façon de rendre explicite l'ordre d'évaluation est d'utiliser l'opérateur de mise en séquence et une structure de boucle.

```
# let imap f l =  
  let l_res = icreate ()  
  in while not (iempty l) do  
    ignore (icons (f (ihd l)) l_res) ;  
    ignore (itl l)  
  done ;  
  { l_res with c = List.rev l_res.c } ;;  
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>  
# let exemple = icons "un" (icons "deux" (icons "trois"  
val exemple : string ilist = {c=["un"; "deux"; "trois"]}  
# imap (function x -> x) exemple ;;  
- : string ilist = {c=["un"; "deux"; "trois"]}
```

La présence de `ignore` illustre bien le fait que ce n'est pas le résultat des fonctions qui nous importe mais leurs effets de bord sur leur argument. Il a de plus été nécessaire de remettre dans le bon ordre (par la fonction `List.rev`) les éléments du résultat.

Récurusif ou itératif

On associe souvent à tort récursif avec fonctionnel et itératif avec impératif. Un programme purement fonctionnel ne pourra pas être itératif car la valeur de la condition d'une boucle ne varie jamais. Par contre un programme impératif peut être récursif : la fonction `imap` en est un exemple.

L'appel d'une fonction conserve les valeurs de ses arguments le temps de son calcul. Si elle appelle une autre fonction, celle-ci conservera en plus ses propres arguments. Ces valeurs sont conservées dans la pile d'exécution. Au retour de l'appel ces valeurs sont dépilées. Cet espace mémoire étant borné, il est possible d'en atteindre la limite en utilisant une fonction récursive dont la profondeur des appels serait trop importante. Dans ce cas Objective CAML déclenche l'exception `Stack_overflow`.

```
# let rec succ n = if n = 0 then 1 else 1 + succ (n-1) ;
```

```
val succ : int -> int = <fun>
# succ 100000 ;;
Stack overflow during evaluation (looping recursion?).
```

Dans la version itérative l'occupation de la pile par l'appel de succ_iter ne dépend pas de son argument.

```
# let succ_iter n =
  let i = ref 0 in
  for j=0 to n do incr i done ;
  !i ;;
val succ_iter : int -> int = <fun>
# succ_iter 100000 ;;
- : int = 100001
```

La version récursive suivante a a priori la même profondeur d'appels et pourtant réussit à s'exécuter avec le même argument.

```
# let succ_rt n =
  let rec succ_aux n accu =
    if n = 0 then accu else succ_aux (n-1) (accu+1)
  in
  succ_aux 1 n ;;
val succ_rt : int -> int = <fun>
# succ_rt 100000 ;;
- : int = 100001
```

Cette fonction a une forme particulière d'appel récursif, dit récursif terminal qui veut que le résultat de l'appel soit le résultat de la fonction sans autre calcul. Il n'est donc pas nécessaire d'avoir conservé les valeurs des arguments de la fonction pendant le calcul de l'appel récursif. Objective CAML quand il sait reconnaître une fonction récursive terminale libère les arguments empilés avant d'effectuer l'appel. Cette optimisation permet d'avoir des fonctions récursives ne faisant pas grossir la pile.

La détection des fonctions récursives terminales existe dans beaucoup de langages, mais elle est indispensable dans un langage fonctionnel où par nature on utilise beaucoup de fonctions récursives terminales.

Quel style choisir ?

Il n'est pas ici question de religion ni d'esthétique, il n'y a pas a priori un style plus joli ou meilleur que l'autre. En revanche, selon le problème à résoudre un style peut être plus adéquat que l'autre.

La première règle à appliquer est celle de la simplicité. L'algorithme à réaliser (qu'il soit écrit dans un livre ou qu'il soit à l'état de germe dans l'esprit du programmeur) est lui-même décrit dans un certain style; il est naturel d'utiliser le même style pour en faire l'implantation.

Le second critère de choix est l'efficacité du programme. On peut dire qu'un programme impératif (bien écrit) est plus efficace que son homologue fonctionnel, mais dans de très nombreux cas la différence n'est pas suffisamment notable pour justifier de compliquer le code en adoptant un style impératif là où le fonctionnel serait naturel. La fonction `map` de la section précédente fournit un bel exemple de problème naturellement exprimé dans le style fonctionnel et qui ne gagne rien à être écrit en impératif.

Séquence ou composition de fonctions

Nous avons vu que dès qu'un programme effectue des effets de bord, il est nécessaire d'explicitier précisément l'ordre d'évaluation des éléments de ce programme. Ceci peut se faire dans les deux styles :

fonctionnel :

en utilisant le fait que Objective CAML est un langage strict, c'est-à-dire que l'argument est calculé avant l'application de la fonction. L'expression `(f (g x))` est calculée en évaluant d'abord `(g x)`, puis en passant le résultat comme argument de `f`. Avec des expressions plus complexes, on peut nommer le résultat intermédiaire avec la construction **let in** mais le principe reste le même : **let** `aux=(g x)` **in** `(f aux)`.

impératif :

en utilisant la séquence ou les autres structures de contrôle (boucles). Dans ce cas, le résultat n'est pas une valeur

rendue par une fonction, mais ses effets de bord sur la mémoire : `aux := (g x) ; (f !aux)`.

Examinons ce problème de choix de style sur un exemple.

L'algorithme de tri rapide sur un vecteur se décrit récursivement comme ceci :

1. choisir un pivot : choisir l'indice d'un élément du tableau;
2. permuter autour du pivot : permuter les éléments du tableau de sorte que les éléments inférieurs au pivot soient situés avant le pivot et vice et versa;
3. trier (suivant le même algorithme) les deux tableaux obtenus : les éléments précédant le pivot et ceux le suivant.

Le choix de l'algorithme, à savoir de modifier un tableau pour que ses éléments soient triés, nous incite à utiliser un style impératif au moins pour les manipulations des données.

On commence par définir une fonction permutant deux éléments d'un tableau.

```
# let permute_element tab n p =  
  let aux = tab.(n) in tab.(n) <- tab.(p) ; tab.(p) <- aux  
val permute_element : 'a array -> int -> int -> unit = <
```

Le choix d'un bon pivot est déterminant pour l'efficacité de l'algorithme, mais nous retenons ici le choix le plus simple : rendre l'indice du premier élément du (sous) tableau.

```
# let choisir_pivot tab debut fin = debut ;;  
val choisir_pivot : 'a -> 'b -> 'c -> 'b = <fun>
```

Écrivons l'algorithme que nous souhaitons utiliser pour permuter les éléments du tableau, autour du pivot.

1. on positionne le pivot au début du tableau en le permutant;
2. *i* est l'indice de second élément du tableau;
3. *j* est l'indice du dernier élément du tableau;
4. si l'élément d'indice *j* est plus grand que le pivot on le permute avec l'élément d'indice *i* et on incrémente *i*; sinon on décrémente *j*;

5. tant que i est inférieur à j on recommence l'opération précédente;
6. rendu à ce point chaque élément d'indice inférieur à i (ou j) est inférieur au pivot, les autres lui sont supérieurs; si l'élément d'indice i est inférieur au pivot on le permute avec le pivot, sinon on permute son prédécesseur.

L'implantation de cet algorithme va naturellement adopter des structures de contrôle impératives.

```
# let permute_pivot tab debut fin ind_pivot =
  permute_element tab debut ind_pivot ;
  let i = ref (debut+1) and j = ref fin and pivot = tab.(
  while !i < !j do
    if tab.(!j) >= pivot then decr j
    else
      begin
        permute_element tab !i !j ;
        incr i
      end
    done ;
    if tab.(!i) > pivot then decr i ;
    permute_element tab debut !i ;
    !i
  ;;
val permute_pivot : 'a array -> int -> int -> int -> int
```

Outre son effet sur le tableau, cette fonction retourne en résultat l'indice du pivot.

Il ne nous reste plus qu'à composer les différentes étapes et mettre en place la récursion sur les sous-tableaux.

```
# let rec quick tab debut fin =
  if debut < fin
  then
    let pivot = choisir_pivot tab debut fin in
    let place_pivot = permute_pivot tab debut fin pivot in
    quick (quick tab debut (place_pivot-1)) (place_pivot+1)
  else tab ;;
```

```
val quick : 'a array -> int -> int -> 'a array = <fun>
```

Nous avons ici utilisé les deux styles. Le pivot choisi sert d'argument à la permutation autour de ce pivot, et le rang du pivot obtenu après permutation est un argument de l'appel récursif. Par contre, le tableau obtenu après permutation n'est pas retourné par la fonction `permute_pivot`, ce résultat est rendu par effet de bord. En revanche, la fonction `quick` retourne un tableau et le tri des sous-tableaux est obtenu par composition des appels récursifs.

La fonction principale est :

```
# let quicksort tab = quick tab 0 ((Array.length tab)-1)
val quicksort : 'a array -> 'a array = <fun>
```

C'est une fonction polymorphe car la relation d'ordre `<` sur les éléments du tableau est elle même polymorphe.

```
# let t1 = [|4;8;1;12;7;3;1;9|] ;;
val t1 : int array = [|4; 8; 1; 12; 7; 3; 1; 9|]
# quicksort t1 ;;
- : int array = [|1; 1; 3; 4; 7; 8; 9; 12|]
# t1 ;;
- : int array = [|1; 1; 3; 4; 7; 8; 9; 12|]
# let t2 = [|"le"; "petit"; "chat"; "est"; "mort"|] ;;
val t2 : string array = [|"le"; "petit"; "chat"; "est";
# quicksort t2 ;;
- : string array = [|"chat"; "est"; "le"; "mort"; "petit
# t2 ;;
- : string array = [|"chat"; "est"; "le"; "mort"; "petit
```

Partage ou copie de valeurs

Tant que les valeurs que nous manipulons ne sont pas modifiables, il n'est pas utile de savoir si elles sont partagées ou non.

```
# let id x = x ;;
```

```

val id : 'a -> 'a = <fun>
# let a = [ 1; 2; 3 ] ;;
val a : int list = [1; 2; 3]
# let b = id a ;;
val b : int list = [1; 2; 3]

```

Que b soit une copie de la liste a ou que se soit la même liste ne change rien puisque ce sont de toutes façons des valeurs intangibles. Mais si à la place d'entiers nous mettons des valeurs modifiables, nous avons besoin de savoir si la modification d'une valeur entraîne la modification de l'autre.

L'implantation du polymorphisme d'Objective CAML effectue une copie des valeurs immédiates et le partage des valeurs structurées. Bien que le passage des arguments soit toujours par valeur, seul le pointeur d'une valeur structurée est copié. C'est bien le cas de la fonction id :

```

# let a = [| 1 ; 2 ; 3 |] ;;
val a : int array = [|1; 2; 3|]
# let b = id a ;;
val b : int array = [|1; 2; 3|]
# a.(1) <- 4 ;;
- : unit = ()
# a ;;
- : int array = [|1; 4; 3|]
# b ;;
- : int array = [|1; 4; 3|]

```

Nous avons ici un vrai choix de programmation pour décider quelle est la manière la plus efficace de représenter une donnée. D'un côté, l'utilisation de valeurs modifiables permet de faire des manipulations en place (c'est-à-dire sans faire d'allocation) mais oblige, dans certains cas, à faire des copies là où l'utilisation d'une valeur non modifiable aurait permis de faire un partage. Nous illustrons ceci avec deux façons d'implanter les listes.

```

# type 'a liste_fixe = LFnill | LFcons of 'a * 'a liste_fixe
# type 'a liste_modifiable = LMnill | LMcons of 'a * 'a liste_modifiable

```

Les listes fixes sont strictement équivalentes aux listes d'Objective CAML alors que les listes modifiables sont plus dans un style à la C où un maillon est une valeur plus une référence sur le maillon suivant.

Avec les listes fixes, il n'y a qu'une seule façon d'écrire la concaténation et elle impose de dupliquer la structure de la première liste; en revanche, la seconde liste peut être partagée avec le résultat.

```
# let rec concat l1 l2 = match l1 with
  LFnil -> l2
  | LFcons (a,l11) -> LFcons(a, (concat l11 l2)) ;;
val concat : 'a liste_fixe -> 'a liste_fixe -> 'a liste_fixe

# let lf1 = LFcons(1, LFcons(2, LFnil))
  and lf2 = LFcons(3, LFcons(4, LFnil)) ;;
val lf1 : int liste_fixe = LFcons (1, LFcons (2, LFnil))
val lf2 : int liste_fixe = LFcons (3, LFcons (4, LFnil))
# let lf3 = concat lf1 lf2 ;;
val lf3 : int liste_fixe =
  LFcons (1, LFcons (2, LFcons (3, LFcons (4, LFnil))))
# lf1==lf3 ;;
- : bool = false
# let tllf l = match l with
  LFnil -> failwith "Liste vide"
  | LFcons (_,x) -> x ;;
val tllf : 'a liste_fixe -> 'a liste_fixe = <fun>
# tllf(tllf(lf3)) == lf2 ;;
- : bool = true
```

Grâce à ces exemples, nous voyons que les premiers maillons de lf1 et de lf3 sont distincts alors que la seconde partie de lf3 est exactement lf2.

Avec les listes modifiables, nous avons le choix entre modifier les arguments (fonction concat_share) ou créer une nouvelle valeur (fonction concat_copy)

```
# let rec concat_copy l1 l2 = match l1 with
```

```

LMnil -> l2
| LMcons (x,l11) -> LMcons(x, ref (concat_copy !l11 l2))
val concat_copy :
  'a liste_modifiable -> 'a liste_modifiable -> 'a liste_

```

Cette première solution, `concat_copy`, donne un résultat similaire à la fonction précédente `concat`. Voici une seconde solution qui partage entièrement arguments et résultat.

```

# let concat_share l1 l2 =
  match l1 with
  | LMnil -> l2
  | _ -> let rec set_last = function
    LMnil -> failwith "concat_share : cas impossible"
    | LMcons (_,l) -> if !l=LMnil then l:=l2 else set_last !
  in
    set_last l1 ;
  l1 ;;
val concat_share :
  'a liste_modifiable -> 'a liste_modifiable -> 'a liste_

```

La concaténation avec partage ne demande aucune allocation (on n'utilise pas le constructeur `LMcons`), on se contente de faire pointer le dernier maillon de la première liste vers la seconde. Mais alors, la concaténation est susceptible de modifier les arguments qui sont passés à la fonction.

```

# let lm1 = LMcons(1, ref (LMcons(2, ref LMnil)))
  and lm2 = LMcons(3, ref (LMcons(4, ref LMnil))) ;;
val lm1 : int liste_modifiable =
  LMcons (1, {contents=LMcons (2, {contents=LMnil})})
val lm2 : int liste_modifiable =
  LMcons (3, {contents=LMcons (4, {contents=LMnil})})
# let lm3 = concat_share lm1 lm2 ;;
val lm3 : int liste_modifiable =
  LMcons (1, {contents=LMcons (2, {contents=LMcons (...))})

```

On obtient bien le résultat escompté pour `lm3`. Par contre la valeur

liée à `lm1` été modifiée.

```
# lm1 ;;  
- : int liste_modifiable =  
LMcons (1, {contents=LMcons (2, {contents=LMcons (...))})
```

Cela peut donc avoir des conséquences dans le reste du programme.

Éléments de choix

Dans un programme purement fonctionnel, il est interdit d'effectuer des effets de bord, cela prohibe la manipulation de structures de données mutables, les exceptions et les entrées-sorties. Nous donnons une définition moins restrictive du style fonctionnel en disant qu'une fonction ne modifiant pas son environnement global peut être utilisée dans un style fonctionnel. Une telle fonction peut manipuler localement des valeurs mutables (et donc être écrite dans un style impératif) mais ne doit pas modifier de variables globales ni ses arguments. Nous les autorisons de surcroît à lever des exceptions. Vu de l'extérieur, ces fonctions peuvent être considérées comme des « boîtes noires » dont le comportement est assimilable à celui d'une fonction purement fonctionnelle si ce n'est qu'elles sont susceptibles de rompre le contrôle par le déclenchement d'une exception. Dans le même esprit, une valeur mutable qui n'est plus modifiée après son initialisation peut être utilisée dans un style fonctionnel.

D'un autre côté, un programme écrit dans un style impératif bénéficie tout de même des avantages apportés par Objective CAML : la sûreté du typage statique, la gestion automatique de la mémoire, le mécanisme d'exception, le polymorphisme paramétrique et l'inférence de types.

Le choix entre style impératif et style fonctionnel dépend de l'application à développer. On peut néanmoins indiquer certaines orientations selon les caractéristiques de celle-ci et les critères jugés important dans son développement.

- **choix des structures de données** : de l'utilisation de

structures de données mutables ou non va découler le choix du style de programmation à adopter. En effet, le style fonctionnel est par nature incompatible avec la modification des valeurs mutables. Par contre, la construction et le parcours d'une valeur sont les mêmes quelque soit son statut. Nous rejoignons la discussion sur < < modification en place vs copie > > tenue à la page ???. Nous y reviendrons en évoquant les critères d'efficacité.

- **structures de données imposées** : si le programme doit modifier des données mutables, le style impératif est le seul possible. Si par contre, il est juste nécessaire de parcourir les valeurs, alors l'adoption du style fonctionnel garantit l'intégrité des données.

L'utilisation de structures de données récursives entraînent l'usage de fonctions elles-mêmes récursives. Celles-ci peuvent être définies en utilisant les traits de l'un ou de l'autre des styles, mais il est souvent plus facile de raisonner par création d'une valeur suivant une définition par récurrence ce qui correspond à une approche fonctionnelle, que d'itérer des traitements par récursion sur cette valeur. Le style fonctionnel permet de définir des itérateurs génériques sur la structure de données manipulée, ce qui factorise le développement et le rend plus rapide.

- **critères d'efficacité** : la modification en place est incomparablement plus efficace que la création d'une valeur. Quand l'efficacité du code est un critère prépondérant, il fera le plus souvent penché la balance vers le style impératif. Notons cependant que la nécessité de départager les valeurs peut s'avérer une tâche très ardue et en définitive plus coûteuse que de copier les valeurs dès le départ. La pleine fonctionnalité a un coût : l'application partielle et l'utilisation de fonctions passées en argument d'autres fonctions ont un coût à l'exécution plus important que celui de l'application totale d'une fonction provenant d'une déclaration. L'usage de ce trait éminemment fonctionnel doit donc être proscrit des parties d'un programme où l'efficacité est critique.

- **critères de développement** : le plus haut niveau

d'abstraction de la programmation fonctionnelle permet d'écrire plus rapidement, un code plus compact et contenant moins d'erreurs que son équivalent impératif qui par nature est plus verbeux. Le style fonctionnel apparaît comme avantage pour les contraintes que posent le développement d'applications conséquentes. L'indépendance d'une fonction par rapport à son contexte d'évaluation rend le code fonctionnel plus facilement découposable en petites unités examinables séparément et en fin de compte plus lisible. La plus grande modularité du style fonctionnel avec la possibilité de passer des fonctions (donc des traitements) en argument d'autres fonctions fait que les programmes écrits dans ce style sont plus aisément réutilisables.

Ces quelques remarques montrent qu'il est souvent judicieux de mélanger les deux styles de programmation dans une application. Le style fonctionnel est plus rapide à développer et confère une organisation plus simple de l'application; mais les sections critiques en temps d'exécution auront intérêt à être développée dans un style impératif plus efficace.

Mélange des styles

Comme nous l'avons déjà mentionné, un langage offrant à la fois des traits fonctionnels et impératifs permet de pouvoir choisir le style le plus approprié pour chaque partie de l'implantation d'un algorithme. On peut tout aussi bien utiliser conjointement les deux aspects du langage dans une même fonction. C'est ce que nous illustrons dans ce paragraphe.

Fermetures et effets de bord

Il est d'usage, quand une fonction effectue un effet de bord, de la traiter comme une procédure et de retourner la valeur (), de type *unit*. Néanmoins, dans certains cas, il peut être utile d'effectuer l'effet de bord à l'intérieur d'une telle fonction et de retourner une valeur pertinente. Nous avons déjà utilisé ce mélange des styles dans la fonction `permuter_pivot` du tri rapide.

L'exemple suivant est un générateur de symboles qui permet d'engendrer un nouveau symbole à chaque appel de la fonction. On utilise simplement un compteur qui est incrémenté à chaque appel.

```
# let c = ref 0;;
val c : int ref = {contents=0}
# let reset_symb = function () -> c:=0 ;;
val reset_symb : unit -> unit = <fun>
# let new_symb = function s -> c:=!c+1 ; s^(string_of_int !c)
val new_symb : string -> string = <fun>
# new_symb "VAR" ;;
- : string = "VAR1"
# new_symb "VAR" ;;
- : string = "VAR2"
# reset_symb () ;;
- : unit = ()
# new_symb "WAR" ;;
- : string = "WAR1"
# new_symb "WAR" ;;
- : string = "WAR2"
```

On peut cacher au reste du programme la référence c en écrivant :

```
# let (reset_s , new_s) =
  let c = ref 0
  in let f1 () = c := 0
     and f2 s = c := !c+1 ; s^(string_of_int !c)
     in (f1,f2) ;;
val reset_s : unit -> unit = <fun>
val new_s : string -> string = <fun>
```

Cette déclaration crée un couple de fonctions qui partagent toutes deux la variable locale (à la déclaration) c. L'utilisation de ces deux fonctions produit la même exécution que les définitions précédentes.

```
# new_s "VAR";;
- : string = "VAR1"
# new_s "VAR";;
- : string = "VAR2"
# reset_s();;
```

```

- : unit = ()
# new_s "WAR";;
- : string = "WAR1"
# new_s "WAR";;
- : string = "WAR2"

```

Cet exemple permet d'illustrer la représentation des fermetures. Une fermeture peut être considérée comme un couple contenant d'une part le code (c'est à dire la partie **function**) et d'autre part un environnement local, contenant les valeurs des variables libres de la fermeture. La figure 4.1 montre la représentation mémoire des fermetures `reset_s` et `new_s`.

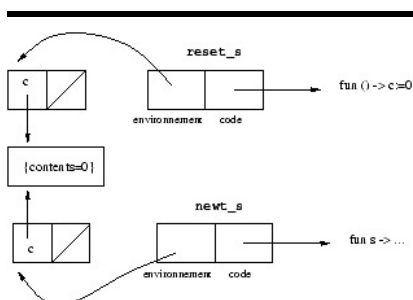


Figure 4.1 : Représentation mémoire de fermetures

Ces deux fermetures partagent le même environnement (la valeur de `c`). Quand l'une modifie la référence `c`, elle modifie le contenu d'une zone mémoire partagée par l'autre fermeture.

Modifications physiques et exceptions

Les exceptions permettent de récupérer des situations où le calcul ne peut pas se poursuivre. Dans ces cas, un récupérateur d'exceptions permet de continuer le calcul sachant qu'une des branches a échoué. Le problème avec les effets de bord provient alors de l'état des données modifiables quand l'exception se déclenche. On ne peut pas garantir cet état s'il y a eu des

modifications physiques dans la branche de calcul qui échoue.

Définissons la fonction d'incrément (`++`) fonctionnant à l'instar de l'opérateur C :

```
# let (++) x = x:=!x+1; x;;  
val ++ : int ref -> int ref = <fun>
```

L'exemple suivant montre un petit calcul où une division par zéro intervient en concomitance avec un effet de bord :

```
# let x = ref 2;;  
val x : int ref = {contents=2}  
(* 1 *)  
# !((++) x) * (1/0) ;;  
Uncaught exception: Division_by_zero  
# x;;  
- : int ref = {contents=2}  
(* 2 *)  
# (1/0) * !((++) x) ;;  
Uncaught exception: Division_by_zero  
# x;;  
- : int ref = {contents=3}
```

La variable `x` n'est pas modifiée lors du calcul de l'expression en `(*1*)`, par contre elle l'est pour l'expression en `(*2*)`. Sauf à savoir sauver les valeurs initiales, les constructions **try .. with ..** ne doivent pas (dans la partie **with ..**) tenir compte des variables modifiables impliquées dans l'expression ayant déclenché une exception.

Structures de données fonctionnelles modifiables

Une manière de voir qu'en programmation fonctionnelle un programme, au sens d'une expression fonctionnelle, est aussi une donnée manipulable, est d'écrire les listes d'association sous forme d'expressions fonctionnelles. On peut en effet voir, et implanter, une liste d'association (`'a * 'b`) *list* comme une fonction partielle de `'a` (l'ensemble des clés) dans `'b` (l'ensemble des valeurs associées).

Autrement dit, une fonction de type $a \rightarrow b$.

La liste vide est la fonction non définie que l'on simule par le déclenchement d'une exception :

```
# let nil_assoc = function x -> raise Not_found ;;
val nil_assoc : 'a -> 'b = <fun>
```

On peut alors écrire la fonction `add_assoc` qui ajoute un élément à une liste, c'est-à-dire qui étend la fonction pour une nouvelle entrée :

```
# let add_assoc (k,v) l = function x -> if x = k then v
val add_assoc : 'a * 'b -> ('a -> 'b) -> 'a -> 'b = <fun>
# let l = add_assoc ('1', 1) (add_assoc ('2', 2) nil_assoc)
val l : char -> int = <fun>
# l '2' ;;
- : int = 2
# l 'x' ;;
Uncaught exception: Not_found
```

On peut réécrire la fonction `mem_assoc` :

```
# let mem_assoc k l = try (l k) ; true with Not_found -> false
val mem_assoc : 'a -> ('a -> 'b) -> bool = <fun>
# mem_assoc '2' l ;;
- : bool = true
# mem_assoc 'x' l ;;
- : bool = false
```

En revanche, il n'est pas immédiat d'écrire une fonction retirant un élément de la liste car on n'a plus accès aux valeurs capturées par les fermetures. Pour cela on masquera l'ancienne valeur par le déclenchement de l'exception `Not_found`.

```
# let rem_assoc k l = function x -> if x=k then raise Not_found else l x
val rem_assoc : 'a -> ('a -> 'b) -> 'a -> 'b = <fun>
# let l = rem_assoc '2' l ;;
val l : char -> int = <fun>
# l '2' ;;
Uncaught exception: Not_found
```

On peut, bien évidemment, créer des références et travailler par effet de bord sur de telles valeurs. Il faut cependant prendre quelques précautions.

```
# let add_assoc_bis (k,v) l = l := (function x -> if x=
val add_assoc_bis : 'a * 'b -> ('a -> 'b) ref -> unit =
```

On obtient pour `l` une fonction qui pointera sur elle-même et donc bouclera. Ce fâcheux effet de bord est dû au fait que le déréréférencement `!l` est dans la portée de la fermeture **function** `x -> .` La valeur de `!l` n'est pas évaluée à la compilation, mais à l'exécution. À ce moment là, `l` pointe sur la valeur modifiée par `add_assoc`. Il faut donc amender notre définition en utilisant la fermeture créée par notre définition primitive de `add_assoc` :

```
# let add_assoc_bis (k, v) l = l := add_assoc (k, v) !l
val add_assoc_bis : 'a * 'b -> ('a -> 'b) ref -> unit =
# let l = ref nil_assoc ;;
val l : ('_a -> '_b) ref = {contents=<fun>}
# add_assoc_bis ('1',1) l ;;
- : unit = ()
# add_assoc_bis ('2',2) l ;;
- : unit = ()
# !l '1' ;;
- : int = 1
# !l 'x' ;;
Uncaught exception: Not_found
```

Structures paresseuses modifiables

L'utilisation de traits impératifs combinés avec un langage fonctionnel procure de bons outils d'implantation de langages informatiques. Nous proposons dans ce paragraphe d'illustrer ce trait par l'implantation de structures de données à évaluation retardée. Une telle structure de données n'est pas complètement évaluée. Son évaluation avance selon l'usage qui en est fait.

Pour simuler l'évaluation retardée, souvent utilisée dans les

langages fonctionnels purs, on utilise des valeurs fonctionnelles, possiblement modifiables. L'intérêt de manipuler des données non complètement évaluées est au moins double : premièrement, ne calculer que ce qui est effectivement utile au calcul ; deuxièmement, travailler sur des données potentiellement infinies.

On définit le type `vm` qui permet d'obtenir soit une valeur déjà calculée (constructeur `Imm`), soit une valeur à calculer (constructeur `Ret`) :

```
# type 'a vm =  
  Imm of 'a  
  | Ret of (unit -> 'a) ;;  
# type 'a vm = {mutable c : 'a vm};;
```

Le retardement du calcul est obtenu par encapsulation dans une fermeture. La fonction d'évaluation d'une telle valeur doit soit retourner une valeur déjà calculée, soit évaluer puis stocker une valeur non encore calculée.

```
# let eval e = match e.c with  
  Imm a -> a  
  | Ret f -> let u = f () in e.c <- Imm u ; u ;;  
val eval : 'a vm -> 'a = <fun>
```

Les opérations de retardement de l'évaluation et d'activation de cette évaluation s'appellent aussi geler et dégeler une valeur.

On peut ainsi écrire la structure conditionnelle sous forme de fonction :

```
# let si_ret c e1 e2 =  
  if eval c then eval e1 else eval e2;;  
val si_ret : bool vm -> 'a vm -> 'a vm -> 'a = <fun>
```

Voici comment on l'utilise pour une fonction récursive comme la factorielle :

```
# let rec facr n =  
  si_ret {c=Ret(fun () -> n = 0)}  
  {c=Ret(fun () -> 1)}
```



```

  {c=Ret (fun () -> n*(facr(n-1))) };;
val facr : int -> int = <fun>
# facr 5;;
- : int = 120

```

Notons que la forme classique du **if** ne peut pas être écrite sous forme de fonction. En effet si l'on définit une fonction si de la manière suivante :

```

# let si c e1 e2 = if c then e1 else e2;;
val si : bool -> 'a -> 'a -> 'a = <fun>

```

Alors les trois arguments de si sont évalués lors de leur passage. Ce qui a pour conséquence que la fonction fact boucle car l'appel récursif fact(n-1) est évalué dans tous les cas, y compris dans le cas où n vaut 0.

```

# let rec fact n = si (n=0) 1 (n*fact(n-1)) ;;
val fact : int -> int = <fun>
# fact 5 ;;
Stack overflow during evaluation (looping recursion?).

```

Module Lazy

La difficulté d'implantation des valeurs gelées provient de la construction d'expressions non évaluées dans le contexte de l'évaluation immédiate d'Objective CAML. Notre tentative de redéfinir la conditionnelle l'a montré. Plus généralement, il n'est pas possible d'écrire une fonction qui gèle une valeur en construisant un objet de type *vm* :

```

# let gele e = { c = Ret (fun () -> e) };;
val gele : 'a -> 'a vm = <fun>

```

Cette fonction suit la stratégie d'évaluation d'Objective CAML et donc évalue l'expression e passée en argument avant de construire la fermeture **fun** () -> e. On le voit sur l'exemple suivant :

```

# gele (print_string "trace"; print_newline(); 4*5);;

```

```
trace
- : int vm = {c=Ret <fun>}
```

Pour cette raison, la forme syntaxique suivante a été introduite.

Syntaxe

lazyexpr

Warning

Ce trait fait partie des extensions du langage et pourra évoluer dans les prochaines versions. Le mot clé **lazy** appliqué à une expression construit une valeur d'un type particulier déclaré dans le module Lazy :

```
# let x = lazy (print_string "Hello"; 3*4) ;;
val x : int Lazy.status ref = {contents=Lazy.Delayed <fu
```

L'expression (print_string "Hello") n'a pas été évaluée puisqu'aucun affichage n'a eu lieu. On peut forcer cette évaluation par appel à la fonction force du module Lazy :

```
# Lazy.force x ;;
Hello- : int = 12
```

On remarque qu'alors la valeur de x a changé :

```
# x ;;
- : int Lazy.t = {contents=Lazy.Value 12}
```

C'est devenu la valeur de l'expression gelée, ici : 12.

Un nouvel appel à la fonction force se contente alors de retourner la valeur calculée :

```
# Lazy.force x ;;
- : int = 12
```

La chaîne "Hello" n'est plus affichée.

Structures de données « infinies »

Le deuxième intérêt de l'évaluation retardée est de construire des structures de données potentiellement infinies tel l'ensemble des entiers naturels. Comme il risquerait d'être long de tous les calculer, l'idée ici est de ne calculer qu'une tête et de savoir passer à l'élément suivant.

On définit une structure générique *'a enum* qui permettra d'énumérer les éléments d'un ensemble.

```
# type 'a enum = { mutable i : 'a; f : 'a -> 'a } ;;
type 'a enum = { mutable i: 'a; f: 'a -> 'a }
# let next e = let x = e.i in e.i <- (e.f e.i) ; x ;;
val next : 'a enum -> 'a = <fun>
```

On peut ainsi obtenir l'ensemble des entiers naturels par instantiation des champs de cette structure :

```
# let nat = { i=0; f=fun x -> x + 1 };;
val nat : int enum = {i=0; f=<fun>}
# next nat;;
- : int = 0
# next nat;;
- : int = 1
# next nat;;
- : int = 2
```

Un autre exemple est l'ensemble des éléments de la suite de Fibonacci dont la définition est :

```
î
í
î
```

La fonction de calcul du successeur doit tenir compte de la valeur courante ($un-1$), mais aussi de la précédente ($un-2$). C'est à cela que sert l'état c de la fermeture.

```
# let fib = let fx = let c = ref 0 in fun v -> let r = !
  in { i=1 ; f=fx } ;;
val fib : int enum = {i=1; f=<fun>}
# for i=0 to 10 do print_int (next fib); print_string "
1 1 2 3 5 8 13 21 34 55 89 - : unit = ()
```

Flots de données

Les streams, ou flots, sont des séquences, potentiellement infinies, d'éléments de même nature. L'évaluation d'une partie d'un flot s'effectue à la demande, c'est-à-dire quand cela est nécessaire pour le calcul en cours : c'est une structure de données paresseuse.

Le type *stream* est un type de données abstrait dont on ne connaît pas l'implantation. On manipule les objets de ce type par des fonctions de construction et de destruction. Pour le confort de l'utilisateur, Objective CAML fournit des constructions syntaxiques simples pour construire des flots et accéder à leurs éléments.

Warning

Les streams sont une extension du langage et ne font pas partie du noyau stable d'Objective CAML.

Construction

La syntaxe allégée de construction des flots est inspirée de celle des liste ou des tableaux. Le flot vide s'écrit :

```
# [< >] ;;
- : 'a Stream.t = <abstr>
```

On peut construire un flot par énumération de ses éléments en les faisant précéder d'une apostrophe (caractère `'`)

```
# [< '0; '2; '4 >] ;;
- : int Stream.t = <abstr>
```

Les expressions qui ne sont pas précédées d'une apostrophe sont considérées comme des sous-flots :

```
# [< '0; [< '1; '2; '3 >]; '4 >] ;;
- : int Stream.t = <abstr>
# let s1 = [< '1; '2; '3 >] in [< s1; '4 >] ;;
- : int Stream.t = <abstr>
# let concat_stream a b = [< a ; b >] ;;
val concat_stream : 'a Stream.t -> 'a Stream.t -> 'a Stream.t
# concat_stream [< "if"; "c"; "then"; "1" >] [< "else" >]
- : string Stream.t = <abstr>
```

Le module Stream fournit d'autres fonctions de construction. Par exemple, les fonctions `of_channel` et `of_string` retournent un flot contenant une suite de caractères, à partir d'un canal d'entrée ou d'une chaîne de caractères.

```
# Stream.of_channel ;;
- : in_channel -> char Stream.t = <fun>
# Stream.of_string ;;
- : string -> char Stream.t = <fun>
```

Le calcul retardé des flots permet de manipuler des structures de données infinies d'une façon similaire au type *'a enum* défini à la page [??](#). On définit le flot des entiers naturels par son premier élément et une fonction calculant le flot des éléments suivants.

```
# let rec nat_stream n = [< 'n ; nat_stream (n+1) >] ;;
val nat_stream : int -> int Stream.t = <fun>
# let nat = nat_stream 0 ;;
val nat : int Stream.t = <abstr>
```

Destruction et filtrage de flots

La primitive `next` permet à la fois d'évaluer, de récupérer et

d'enlever le premier élément d'un flot :

```
# for i=0 to 10 do
  print_int (Stream.next nat) ;
  print_string " "
done ;;
0 1 2 3 4 5 6 7 8 9 10 - : unit = ()
# Stream.next nat ;;
- : int = 11
```

Lorsque le flot est épuisé, une exception est déclenchée.

```
# Stream.next [< >] ;;
Uncaught exception: Stream.Failure
```

Pour manipuler les flots, Objective CAML offre une construction de filtrage dédiée dit filtrage destructif. La valeur filtrée est calculée et retirée du flot. Il n'y a pas de notion d'exhaustivité du filtrage des flots et, parce que nous manipulons une structure paresseuse potentiellement infinie, on peut ne pas filtrer la totalité du flot. La syntaxe de filtrage est :

Syntaxe

matchexprwith parser [*<p1 ...>*]->*expr1* | ...

La fonction next peut s'écrire :

```
# let next s = match s with parser [< 'x >] -> x ;;
val next : 'a Stream.t -> 'a = <fun>
# next nat;;
- : int = 12
```

Remarquez que l'énumération des entiers reprend là où nous l'avions laissée.

À la manière de l'abstraction, il existe une forme syntaxique filtrant un paramètre de type *Stream.t*.

Syntaxe

parserp -> ...

La fonction **next** peut alors se réécrire ainsi :

```
# let next = parser [<'x>] -> x ;;
val next : 'a Stream.t -> 'a = <fun>
# next nat ;;
- : int = 13
```

Il est possible de filtrer le flot vide, mais attention : le motif de flot [**<>**] filtre n'importe quel flot. En effet, un flot **s** est toujours égal au flot [**<>**]; **s** >]. Il faut en conséquence inverser l'ordre usuel du filtrage :

```
# let rec it_stream f s =
  match s with parser
  | [<'x ; ss >] -> f x ; it_stream f ss
  | [<>] -> () ;;
val it_stream : ('a -> 'b) -> 'a Stream.t -> unit = <fun>
# let print_int1 n = print_int n ; print_string" " ;;
val print_int1 : int -> unit = <fun>
# it_stream print_int1 [<'1; '2; '3>] ;;
1 2 3 - : unit = ()
```

En utilisant le fait que le filtrage est destructif, on peut également écrire :

```
# let rec it_stream f s =
  match s with parser
  | [<'x >] -> f x ; it_stream f s
  | [<>] -> () ;;
val it_stream : ('a -> 'b) -> 'a Stream.t -> unit = <fun>
# it_stream print_int1 [<'1; '2; '3>] ;;
1 2 3 - : unit = ()
```

Si les flots sont paresseux, ils sont cependant de bonne volonté et ne refusent jamais de fournir leur premier élément et lorsque celui-ci est fourni, il est perdu. Ceci a des conséquences sur le filtrage. La

fonction suivante est une tentative (vouée à l'échec) d'un affichage par couple d'un flot d'entiers, sauf éventuellement pour le dernier élément.

```
# let print_int2 n1 n2 =  
  print_string "(" ; print_int n1 ; print_string "," ;  
  print_int n2 ; print_string ")" ;;  
val print_int2 : int -> int -> unit = <fun>  
# let rec print_stream s =  
  match s with parser  
  | [< 'x; 'y >] -> print_int2 x y; print_stream s  
  | [< 'z >] -> print_int1 z; print_stream s  
  | [<>] -> print_newline() ;;  
val print_stream : int Stream.t -> unit = <fun>  
# print_stream [<'1; '2; '3>];;  
(1,2)Uncaught exception: Stream.Error("")
```

Les deux premiers éléments du flot ont bien été affichés, mais lors de l'évaluation de l'appel récursif (`print_stream [< 3 >]`) le premier motif a trouvé une valeur pour `x`, qui a donc été consommée, mais il ne restait plus rien pour `y`. C'est ce qui a provoqué l'erreur. En fait, le second filtre est inutile puisque, si le flot n'est pas vide, le premier motif peut toujours commencer l'évaluation.

Pour obtenir le résultat attendu, il faut séquentialiser le filtrage :

```
# let rec print_stream s =  
  match s with parser  
  | [< 'x >]  
  -> (match s with parser  
  | [< 'y >] -> print_int2 x y; print_stream s  
  | [<>] -> print_int1 x; print_stream s)  
  | [<>] -> print_newline() ;;  
val print_stream : int Stream.t -> unit = <fun>  
# print_stream [<'1; '2; '3>];;  
(1,2)3  
- : unit = ()
```

Si le filtrage échoue sur le premier élément d'un filtre alors, on retrouve le comportement connu du filtrage :


```
# let rec print_stream s =
  match s with parser
  | [< '1; 'y >] -> print_int2 1 y; print_stream s
  | [< 'z >] -> print_int1 z; print_stream s
  | [<>] -> print_newline() ;;
val print_stream : int Stream.t -> unit = <fun>
# print_stream [<'1; '2; '3>] ;;
(1,2)3
- : unit = ()
```

Aux limites du filtrage

Par son caractère destructif, le filtrage des flots diffère du filtrage des types somme. Nous allons voir comment il peut en différer radicalement.

On peut écrire, de façon naturelle, une fonction qui calcule la somme des éléments d'un flot :

```
# let rec somme s =
  match s with parser
  | [< 'n; ss >] -> n+(somme ss)
  | [<>] -> 0 ;;
val somme : int Stream.t -> int = <fun>
# somme [<'1; '2; '3; '4>] ;;
- : int = 10
```

Mais on peut également consommer le flot « de l'intérieur » en nommant le résultat obtenu :

```
# let rec somme s =
  match s with parser
  | [< 'n; r = somme >] -> n+r
  | [<>] -> 0 ;;
val somme : int Stream.t -> int = <fun>
# somme [<'1; '2; '3; '4>] ;;
- : int = 10
```

Nous étudierons d'autres utilisations importantes des flots dans le

chapitre 11 consacré aux analyses lexicale et syntaxique. En particulier, nous verrons comment la consommation d'un flot de l'intérieur peut être mise à profit.

Exercices

Arbres binaires

On représente les arbres binaires sous forme de vecteurs. Soit un arbre a de hauteur h , la taille du vecteur sera $2(h+1)-1$. Soit un noeud en position i , le sous-arbre gauche de ce noeud est dans l'intervalle d'indices $[i+1, i+1+2h]$ et son sous-arbre droit dans l'intervalle $[i+1+2h+1, 2(h+1)-1]$. Cette représentation est intéressante si l'arbre est presque complètement rempli. Les étiquettes de tels arbres doivent posséder une valeur spéciale indiquant que le noeud n'existe pas. On représentera donc des arbres étiquetés par des valeurs de type 'a et par des vecteurs de type 'a array.

1. Écrire une **fonction** qui prend en entrée un **arbre** binaire du type 'a *arbre_bin* défini à la page ??, un tableau, que l'on supposera de taille suffisante, et qui stocke les étiquettes contenues dans l'arbre dans le tableau selon la discipline de placement définie ci-dessus.

```
# let fill_array tree tab empty =  
  let rec aux i p = function  
    Empty -> tab.(i) <- empty  
    | Node (l,e,r) ->  
      tab.(i) <- e ;  
      aux (i+1) (p/2) l ;  
      aux (i+p) (p/2) r  
  in aux 0 (((Array.length tab)+1)/2) tree ;;  
val fill_array : 'a arbre_bin -> 'a array -> 'a ->
```

```
# type 'a arbre_bin =
  Empty
  | Node of 'a arbre_bin * 'a * 'a arbre_bin ;;
type 'a arbre_bin = | Empty | Node of 'a arbre_bin
```

2. Écrire la fonction de création d'une **feuille** (arbre de hauteur 0).

```
# let leaf empty = [| empty |] ;;
val leaf : 'a -> 'a array = <fun>
```

3. Écrire une **fonction** qui construit un nouvel arbre à partir d'une étiquette et de deux autres arbres.

```
# let node elt left right =
  let ll = Array.length left and lr = Array.length right
  let l = max ll lr in
  let res = Array.create (2*l+1) elt in
  Array.blit left 0 res 1 ll ;
  Array.blit right 0 res (ll+1) lr ;
  res ;;
val node : 'a -> 'a array -> 'a array -> 'a array =
```

4. Écrire une **fonction** de conversion du type 'a arbre_bin en un tableau.

```
# let rec make_array empty = function
  Empty -> leaf empty
  | Node (l,e,r) -> node e (make_array empty l) (make_array empty r)
val make_array : 'a -> 'a arbre_bin -> 'a array = <
```

5. Écrire une **fonction** de parcours infixe de tels arbres.

```
# let infixe tab empty f =
  let rec aux i p =
    if tab.(i) <> empty then ( aux (i+1) (p/2) ; f tab.(i) )
    in aux 0 (((Array.length tab)+1)/2) ;;
val infixe : 'a array -> 'a -> ('a -> 'b) -> unit =
```

6. L'utiliser pour l'affichage .

```
# let print_tab_int tab empty =
  infixe tab empty (fun x -> print_int x ; print_str "\n")
val print_tab_int : int array -> int -> unit = <fun
```

7. Que dire du parcours [préfixe](#) de tels arbres?

```
# let prefixe tab empty f =
  for i=0 to (Array.length tab)-1 do if tab.(i) <> empty then f tab.(i)
val prefixe : 'a array -> 'a -> ('a -> unit) -> unit =
```

Correcteur orthographique

Cet exercice reprend les [arbres lexicaux](#), exercice du chapitre 2, page ??, pour construire un correcteur orthographique.

1. Construire un dictionnaire à partir d'un fichier en ASCII où chaque ligne contient un mot. On écrira pour cela une [fonction](#) qui prend un nom de fichier et retourne le dictionnaire correspondant.

```
# type noeud_lex = Lettre of char * bool * arbre_lex
and arbre_lex = noeud_lex list;;
type noeud_lex = | Lettre of char * bool * arbre_lex
type arbre_lex = noeud_lex list
# type mot = string;;
```

```

type mot = string
# let rec existe m d =
  let aux sm i n =
    match d with
    [] -> false
    | (Lettre (c,b,l))::q ->
      if c = sm.[i] then
        if n = 1 then b
        else existe (String.sub sm (i+1) (n-1)) l
      else existe sm q
  in aux m 0 (String.length m);;
val existe : string -> arbre_lex -> bool = <fun>

# let rec ajoute m d =
  let aux sm i n =
    if n = 0 then d else
    match d with
    [] -> [Lettre (sm.[i], n = 1, ajoute (String.sub s
    | (Lettre(c,b,l))::q ->
      if c = sm.[i] then
        if n = 1 then (Lettre(c,true,l))::q
        else Lettre(c,b,ajoute (String.sub sm (i+1) (n-1))
        else (Lettre(c,b,l))::(ajoute sm q)
      in aux m 0 (String.length m);;
val ajoute : string -> arbre_lex -> arbre_lex = <fu

# let rec verifie l d = match l with
[] -> []
| t::q -> if existe t d then t::(verifie q d)
else verifie q d
;;
val verifie : string list -> arbre_lex -> string li

# let string_of_char c = String.make 1 c;;
val string_of_char : char -> string = <fun>

# let rec filter p l = match l with
[] -> []
| t::q -> if p t then t::(filter p q)
else filter p q;;

```

```
val filter : ('a -> bool) -> 'a list -> 'a list = <
```

```
# let rec selecte n d =
  match d with
  [] -> []
  | (Lettre(c,b,l))::q ->
    if n = 1 then
      filter (function x -> x <> "!")
      (List.map (function (Lettre(c,b,_)) -> if b then
    else
      let r = selecte (n-1) l
      and r2 = selecte n q in
      let pr = List.map (function s -> (string_of_char c
    in pr@r2;;
val selecte : int -> arbre_lex -> string list = <fu
```

```
# let lire_fichier nom_fichier =
  let dico = ref []
  and canal = open_in nom_fichier in
  try
    while true do dico := ajoute (input_line canal) !dico
    failwith "cas impossible"
  with
    End_of_file -> close_in canal ; !dico
    | x -> close_in canal ; raise x ;;
val lire_fichier : string -> arbre_lex = <fun>
```

2. Écrire une fonction **mots** qui prend une chaîne de caractères et construit la liste des mots de cette chaîne. Les séparateurs de mots sont l'espace, la tabulation, l'apostrophe et les guillemets.

```
# let mots s =
  let est_sep = function ' ' | '\t' | '\'' | '"' -> true |
  let res = ref [] and p = ref ((String.length s)-1)
  let n = ref !p in
```

```

while !p>=0 && est_sep s.[!p] do decr p done ;
n := !p ;
while (!n>=0) do
  while !n>=0 && not (est_sep s.[!n]) do decr n done
  res := String.sub s ( !n +1) (!p - !n) :: !res ;
  while !n>=0 && est_sep s.[!n] do decr n done ;
  p := !n
done ;
!res ;;
val mots : string -> string list = <fun>

```

3. Écrire une fonction **verifie** qui prend un dictionnaire, une liste de mots et retourne la liste des mots qui n'apparaissent pas dans le dictionnaire.

```

# let rec verifie dico = function
  [] -> []
  | m::l -> if existe m dico then verifie dico l else m::l
val verifie : arbre_lex -> string list -> string list

```

4. Écrire une fonction **occurrences** qui prend une liste de mots et retourne une liste de couples associant chaque mot de la liste à son nombre d'occurrences.

```

# let rec ajoute x = function
  [] -> [(x,1)]
  | ((y,n) as p)::l -> if x=y then (y,n+1)::l else p::l
val ajoute : 'a -> ('a * int) list -> ('a * int) list

# let rec ajoute_liste ld = function
  [] -> ld
  | n::l -> let res = ajoute_liste ld l in ajoute n res
val ajoute_liste : ('a * int) list -> 'a list -> ('a * int) list

# let occurrences l = ajoute_liste [] l ;;
val occurrences : 'a list -> ('a * int) list = <fun>

```

5. Écrire une fonction **orthographe** qui prend un dictionnaire, un nom de fichier contenant le texte à analyser, et retourne la liste des mots incorrects avec leur nombre d'apparitions.

```
# let orthographe dico nom =  
  let f = open_in nom and res = ref [] in  
  try  
    while true do  
      let s = input_line f in  
      let ls = mots s in  
      let lv = verifie dico ls in  
      res := ajoute_liste !res lv  
    done ;  
    failwith "cas impossible"  
  with  
  End_of_file -> close_in f ; !res  
  | x -> close_in f ; raise x ;;  
val orthographe : arbre_lex -> string -> (string *
```

Ensemble des nombres premiers

On cherche à construire l'ensemble infini des nombres premiers sans le calculer complètement à la manière des structures de données paresseuses.

1. Définir le **prédicat** **est_divisible** qui prend un entier et une liste initiale de nombres premiers et qui détermine si le nombre est divisible par un des entiers de la liste.

```
# let rec est_divisible x = function  
  [] -> false  
  | n::l -> (x mod n)=0 || ( (n*n<=x) && (est_divis  
val est_divisible : int -> int list -> bool = <fun>
```

2. Possédant une liste initiale de nombres premiers, écrire la

fonction [suivant](#) qui renvoie le plus petit nombre premier qui n'est pas dans cette liste.

```
# let rec dernier = function
  [] -> failwith "liste vide"
  | [x] -> x
  | _::l -> dernier l ;;
val dernier : 'a list -> 'a = <fun>
```

3. Définir la valeur [ensprem](#) représentant l'ensemble des nombres premiers en s'inspirant du type `'a enum` donné page [??](#). Il sera utile pour cet ensemble de conserver les nombres entiers déjà trouvés.

```
# type 'a ens = {mutable i: 'a ; f : 'a -> 'a } ;;
type 'a ens = { mutable i: 'a; f: 'a -> 'a }
# let next e = let x = e.i in e.i <- (e.f e.i) ; x
val next : 'a ens -> 'a = <fun>
```

Résumé

Ce chapitre a comparé les styles de programmation fonctionnel et impératif. Ils diffèrent principalement sur le contrôle de l'exécution (implicite en fonctionnel et explicite en impératif) et sur la représentation mémoire des données (partage ou copie explicite en impératif, non crucial en fonctionnel). Les implantations d'algorithmes dans les styles fonctionnel et impératif tiennent compte de ces différences. Le choix entre ces deux styles amène en fait à leur mélange. Celui-ci permet d'explicitement la représentation des fermetures, d'optimiser des parties cruciales d'applications et de créer des données fonctionnelles modifiables. La modification physique de valeurs dans l'environnement d'une fermeture permet de mieux comprendre ce qu'est une valeur fonctionnelle. Le mélange des deux styles donne des outils d'implantation puissants. Cela a été utilisé pour la construction de valeurs potentiellement infinies.

Pour en savoir plus

Les principales conséquences de l'ajout des traits impératifs dans un langage fonctionnel sont :

- de fixer la stratégie d'évaluation (évaluation stricte);
- d'ajouter des contraintes d'implantation, en particulier pour le GC (voir chapitre 9);
- pour les langages typés statiquement, de complexifier leur système de types;
- d'offrir différents styles de programmation dans le même langage, permettant ainsi l'écriture dans le style approprié ou possiblement mixte d'algorithmes.

Ce dernier point est important en Objective CAML où l'on désire avoir le même polymorphisme paramétrique pour les fonctions écrites dans un style ou dans l'autre. Pour cela, certains programmes purement fonctionnels ne sont plus typables avec cet ajout. L'article de Wright ([[Wri93](#)]) explique les difficultés du polymorphisme dans les langages munis de traits impératifs. Objective CAML adopte la solution qu'il préconise. La classification des différents types de polymorphisme en présence de modification physique est bien décrite dans la thèse d'Emmanuel Engel ([[Eng98](#)]).

Ces conséquences rendent la tâche des programmeurs un peu plus ardue et l'apprentissage du langage un peu plus difficile, mais comme le langage est plus riche et surtout laisse le choix du style, le jeu en vaut la chandelle. Par exemple l'évaluation stricte est de règle, mais il est possible d'implanter les mécanismes de base pour l'évaluation paresseuse grâce au mélange des deux styles. La plupart des langages fonctionnels purs utilisent cette évaluation. Proches de ML on peut citer les langages Miranda, LazyML et Haskell. Les deux premiers sont utilisés à l'université (enseignement et recherche). Par contre, il existe des applications conséquentes écrites en Haskell. L'absence d'effets de bord contrôlables nécessite une abstraction supplémentaire pour les entrées-sorties, appelée monade. On peut lire les ouvrages sur Haskell ([[Tho99](#)]) pour approfondir le sujet. Les flots représentent bien ce mélange de style fonctionnel et impératif. Leur utilisation pour les analyses lexicale et syntaxique est détaillée au chapitre 11.

Interface graphique

Introduction

Ce chapitre présente la bibliothèque Graphics fournie avec la distribution du langage Objective CAML. Cette bibliothèque fonctionne de manière identique sur les principales interfaces graphiques des systèmes d'exploitation les plus courants : Windows, MacOS, Unix avec X-Windows. Graphics permet la réalisation de tracés graphiques, pouvant contenir du texte et des images, et gère des événements de base comme le clic de la souris ou l'appui d'une touche du clavier.

Le modèle de programmation pour les tracés graphiques est le `<< modèle du peintre >>` : la dernière couche de peinture efface la précédente. C'est un modèle impératif dans le sens où la fenêtre graphique est un tableau de points que chaque primitive graphique modifie physiquement. Les interactions avec la souris et le clavier seront le cadre de la programmation par événements : la fonction principale du programme est une boucle sans fin en attente d'une action de l'utilisateur. La venue d'un événement entraîne l'exécution d'un traitement particulier, puis le retour à la boucle principale en attente d'un prochain événement.

Bien que la bibliothèque Graphics soit très simple, elle est suffisante d'une part pour introduire les concepts de base des interfaces graphiques, et d'autre part, elle contient les éléments de base pour l'élaboration d'interfaces graphiques plus riches et plus faciles à manipuler par le programmeur.

Plan du chapitre

La première section explique comment se servir de cette bibliothèque sur les différents systèmes. La deuxième section introduit les notions de base de la programmation graphique : repère, tracé, remplissage, couleurs, bitmaps. La troisième section illustre ces concepts en décrivant et implantant des fonctions de création et de dessin de « boîtes ». La quatrième section montre l'animation d'objets graphiques et leur interaction avec le fond

d'écran ou les autres objets animés. La cinquième section présente le style de la programmation par événements ainsi que le squelette de toute interface graphique. Enfin la dernière section utilise la bibliothèque Graphics pour construire une interface graphique pour la calculatrice donnée page [??](#).

Mise en oeuvre du module

L'utilisation de la bibliothèque Graphics diffère selon le système et le mode de compilation utilisés. On ne s'intéressera ici qu'aux applications utilisables à partir de la boucle d'interaction d'Objective CAML. Pour les systèmes Windows ou MacOS, l'environnement de travail interactif a déjà chargé la bibliothèque. Pour les systèmes Unix, il est nécessaire de construire un nouveau toplevel. Celui-ci dépend de l'emplacement de la bibliothèque X11. Si celle-ci est placée dans un des chemins habituels de recherche des bibliothèques du langage C, alors la ligne de commande est la suivante :

```
ocamlmktop -custom -o montoplevel graphics.cma -cclib -l
```

Elle construit une nouvelle commande montoplevel où la bibliothèque Graphics est intégrée. Le lancement de la commande s'effectue de la manière suivante :

```
./montoplevel
```

Si par contre, comme sous Linux, la bibliothèque X11 est placée dans un autre catalogue, il faut l'indiquer à la commande ocamlmktop :

```
ocamlmktop -custom -o montoplevel graphics.cma -cclib \
-L/usr/X11/lib -cclib -lX11
```

Dans cet exemple, le fichier libX11.a est recherché dans le catalogue /usr/X11/lib.

Une description complète de la commande ocamlmktop est donnée au chapitre [7](#).

Notions de base

La programmation graphique est intimement liée à l'évolution technologique du matériel, en particulier celle des écrans et des cartes graphiques. Pour que le rendu d'une image soit d'une qualité suffisante, il est nécessaire que le dessin soit rafraîchi (redessiné) à intervalles réguliers et courts, un peu comme au cinéma. Il y a principalement deux techniques pour dessiner à l'écran : soit à partir d'une liste de segments visibles (écran à cavalier) où seule la partie utile du dessin est dessinée, soit en affichant tous les points de l'écran (écran bitmap). C'est cette dernière technologie qui est utilisée sur les micro-ordinateurs du marché.

Les écrans bitmap peuvent être vus comme un rectangle de points accessibles, c'est à dire affichables. On appelle ces points des pixels, pour picture element, en anglais. Ce sont les éléments de base pour construire des images. On appelle définition d'un écran la hauteur et la largeur du bitmap principal. La taille de ce bitmap dépend donc de la taille mémoire de chaque pixel. En noir et blanc, un pixel peut être codé par un bit. Pour les écrans à niveau de gris ou en couleur, la taille d'un pixel dépend alors du nombre de teintes différentes pouvant être données à un pixel. Pour un bitmap de 320x640 pixels avec 256 couleurs par pixel, il sera donc nécessaire de coder un pixel sur 8 bits, ce qui demandera une mémoire vidéo de : $480 * 640 \text{ octets} = 307200 \text{ octets}$

300 ko. Cette résolution est encore utilisée par certains programmes MS-DOS.

Les opérations de base sur un bitmap que l'on retrouve dans la bibliothèque Graphics sont :

- la coloration d'un pixel,
- le tracé d'un segment,
- le tracé d'une forme : rectangle, ellipse,
- le remplissage d'une forme fermée : rectangle, ellipse, polygone,
- l'affichage d'un texte : bitmap ou vectoriel,
- la manipulation ou le déplacement d'une partie de l'image.

Toutes ces opérations s'effectuent dans un repère, celui du bitmap. Un certain nombre de caractéristiques des opérations graphiques, comme la largeur du trait, la jointure des lignes, le choix de la police de caractères, le style et le motif de remplissage définissent ce que l'on appelle un contexte graphique. Une opération graphique s'effectue toujours dans un contexte graphique particulier et son résultat en dépend. Le contexte graphique de la bibliothèque Graphics ne contient que le point courant, la couleur courante, la police courante et sa taille.

Affichage graphique

Les éléments de base de l'affichage graphique sont : le repère et le contexte graphique, les couleurs, les tracés et les remplissages de formes fermées, les textes, et les bitmaps.

Repère et contexte graphique

La bibliothèque Graphics gère une fenêtre principale et unique. Les coordonnées du repère de la fenêtre vont du point (0,0) en bas à gauche au point supérieur droit de la fenêtre. Les principales fonctions sur cette fenêtre sont :

- `open_graph`, de type *string* -> *unit*, qui ouvre une fenêtre;
- `close_graph`, de type *unit* -> *unit*, qui la ferme;
- `clear_graph`, de type *unit* -> *unit*, qui l'efface.

La taille de la fenêtre graphique est donnée par les fonctions `size_x` et `size_y`.

La chaîne de caractères passée en argument de la fonction `open_graph` dépend du système de fenêtrage de la machine sur laquelle est exécuté le programme et n'est donc pas indépendante de la plate-forme. Toutefois, la chaîne vide ouvre une fenêtre dont les caractéristiques sont données par défaut. Il est possible de spécifier la taille de la fenêtre : sous X-Windows, " 200x300 " donnera une fenêtre de 200 pixels de large et de 300 pixels de haut. Attention, l'espace, premier caractère de la chaîne " 200x300 " est indispensable.

Le contexte graphique contient un certain nombre de composants qui sont accessibles ou modifiables :

le point courant :	<code>current_point : unit -> int * int</code>
	<code>moveto : int -> int -> unit</code>
la couleur courante :	<code>set_color : color -> unit</code>
la largeur du tracé de lignes :	<code>set_line_width : int -> unit</code>
la police de caractères courante :	<code>set_font : string -> unit</code>
la taille des caractères :	<code>set_text_size : int -> unit</code>

Couleurs

Les couleurs sont représentées par trois octets : chacun est la valeur de l'intensité d'une couleur principale dans le modèle RGB (rouge, vert, bleu) comprise entre un minimum 0 et un maximum 255. La fonction `rgb` (de type `int -> int -> int -> color`) permet de construire une nouvelle couleur à partir de ses trois composantes. Si les trois composantes sont identiques, la couleur obtenue est un gris plus ou moins clair selon la valeur. Le noir correspond au minimum d'intensité pour chacune des composantes (0 0 0) et le blanc correspond au maximum (255 255 255). Certaines couleurs sont prédéfinies : `black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan`, `magenta`. Les variables `foreground` et `background` correspondent respectivement à la couleur courante et à la couleur du fond. L'effacement de l'écran provoque son remplissage par la couleur du fond.

Une couleur (une valeur de type `color`) est en fait un entier qu'il est possible de manipuler pour, par exemple, la décomposer selon ses trois composantes (`from_rgb`) ou lui appliquer une fonction d'inversion vidéo (`inv_color`).

```
(* couleur == R * 256 * 256 + G * 256 + B *)
# let from_rgb (c : Graphics.color) =
  let r = c / 65536 and g = c / 256 mod 256 and b = c mod 256
  in (r,g,b) ;;
val from_rgb : Graphics.color -> int * int * int = <fun>
# let inv_color (c : Graphics.color) =
```

```

let (r,g,b) = from_rgb c
in Graphics.rgb (255-r) (255-g) (255-b) ;;
val inv_color : Graphics.color -> Graphics.color = <fun>

```

La fonction `point_color`, de type `int -> int -> color`, retourne la couleur du point dont les coordonnées lui ont été passées en argument.

Tracé et remplissage

Une fonction de tracé dessine un trait à l'écran. Le trait est de la largeur courante et de la couleur courante. Une fonction de remplissage colorie une forme fermée avec la couleur courante. Les différentes fonctions de tracé et de remplissage sont données dans la figure 5.1.

tracé	remplissage	type
plot		<code>int -> int -> unit</code>
lineto		<code>int -> int -> unit</code>
	fill_rect	<code>int -> int -> int</code> <code>-> int -> unit</code>
	fill_poly	<code>(int * int) array -</code> <code>> unit</code>
draw_arc	fill_arc	<code>int -> int -> int</code> <code>-> int -> int -></code> <code>unit</code>
draw_ellipse	fill_ellipse	<code>int -> int -> int</code> <code>-> int -> unit</code>
draw_circle	fill_circle	<code>int -> int -> int</code> <code>-> unit</code>

Figure 5.1 : Fonctions de tracé et de remplissage

Attention, la fonction `lineto` a pour effet de bord de modifier la position du point courant.

Tracé de polygones

En guise d'exemple, nous ajoutons les primitives de tracé qui ne sont pas prédéfinies. Un polygone est décrit par un tableau contenant ses sommets.

```
# let draw_rect x0 y0 w h =  
  let (a,b) = Graphics.current_point()  
  and x1 = x0+w and y1 = y0+h  
  in  
    Graphics.moveto x0 y0;  
    Graphics.lineto x0 y1; Graphics.lineto x1 y1;  
    Graphics.lineto x1 y0; Graphics.lineto x0 y0;  
    Graphics.moveto a b ;;  
val draw_rect : int -> int -> int -> int -> unit = <fun>  
  
# let draw_poly r =  
  let (a,b) = Graphics.current_point () in  
  let (x0,y0) = r.(0) in Graphics.moveto x0 y0 ;  
  for i = 1 to (Array.length r)-1 do  
    let (x,y) = r.(i) in Graphics.lineto x y  
  done ;  
  Graphics.lineto x0 y0 ;  
  Graphics.moveto a b ;;  
val draw_poly : (int * int) array -> unit = <fun>
```

Notez que ces fonctions prennent les mêmes arguments que les fonctions prédéfinies de remplissage de ces formes. Et que comme les autres fonctions de tracé de formes, elles ne modifient pas le point courant.

Illustration du modèle du peintre

Cet exemple construit l'illustration d'un réseau à jeton en anneau (figure 5.2). Chaque machine est représentée par un petit cercle. On répartit l'ensemble des machines sur un grand cercle et l'on trace un trait entre les machines connectées. La position courante du jeton dans le réseau est matérialisée par un petit disque noir.

La fonction `ens_points` construit les coordonnées des différentes machines du réseau. Les données résultantes sont stockées dans un tableau.

```
# let pi = 3.1415927 ;;
val pi : float = 3.1415927
# let ens_points (x,y) l n =
  let a = 2. *. pi /. (float n) in
  let rec aux (xa,ya) i =
    if i > n then []
    else
      let na = (float i) *. a in
      let x1 = xa + (int_of_float ( cos(na) *. 1))
      and y1 = ya + (int_of_float ( sin(na) *. 1)) in
      let np = (x1,y1) in
      np::(aux np (i+1))
  in Array.of_list (aux (x,y) 1) ;;
val ens_points : int * int -> float -> int -> (int * int)
```

La fonction `dessine` affiche les connexions, les machines et le jeton.

```
# let dessine (x,y) l n sc st =
  let r = ens_points (x,y) l n in
  draw_poly r ;
  let dessine_cercle (x,y) =
    Graphics.set_color Graphics.background ;
    Graphics.fill_circle x y sc ;
    Graphics.set_color Graphics.foreground ;
    Graphics.draw_circle x y sc
  in
  Array.iter dessine_cercle r ;
  Graphics.fill_circle x y st ;;
val dessine : int * int -> float -> int -> int -> int ->
```

L'appel suivant correspond au dessin gauche de la figure [5.2](#).

```
# dessine (140,20) 60.0 10 10 3;;
- : unit = ()
```

```
# save_screen "IMAGES/tokenring.caa";  
- : unit = ()
```



Figure 5.2 : Réseau en anneau

Notons que l'ordre de l'affichage a son importance. Nous traçons d'abord les connexions puis les noeuds. Le dessin des noeuds du réseau efface une partie des traits de connexion. Il n'y a donc pas besoin de calculer le point d'intersection entre les segments de connexion et les cercles des sommets. L'illustration de droite de la figure 5.2 inverse cet ordre d'affichage. On voit donc le tracé des segments apparaître à l'intérieur des cercles représentant les noeuds.

Texte

Les fonctions d'affichage de texte sont des plus simples. Les deux fonctions `draw_char` (de type *char* -> *unit*) et `draw_string` (de type *string* -> *unit*) affichent respectivement un caractère et une chaîne de caractères au point courant. Après l'affichage, ce dernier est modifié. Ces affichages tiennent compte du choix de la police courante et de sa taille courante.

Remarque

Le rendu d'affichage sur les chaînes peut différer selon les systèmes graphiques.

La fonction `text_size` prend une chaîne en entrée et retourne un couple d'entiers correspondant aux dimensions de cette chaîne affichée avec la police et la taille courante.

Affichage vertical de chaînes

Cet exemple décrit la fonction `draw_string_v` qui affiche une chaîne de caractères verticalement à partir du point courant. Elle est utilisée à la figure 5.3. Chaque lettre est affichée séparément en changeant la coordonnée verticale.

```
# let draw_string_v s =  
  let (xi,yi) = Graphics.current_point()  
  and l = String.length s  
  and (_,h) = Graphics.text_size s  
  in  
    Graphics.draw_char s.[0] ;  
    for i=1 to l-1 do  
      let (_,b) = Graphics.current_point()  
      in Graphics.moveto xi (b-h) ;  
      Graphics.draw_char s.[i]  
    done ;  
    let (a,_) = Graphics.current_point() in Graphics.moveto a xi  
val draw_string_v : string -> unit = <fun>
```

Cette fonction modifie le point courant. Après l'affichage celui-ci se retrouve à la position initiale décalée de la largeur d'un caractère.

Le programme suivant permet d'afficher une légende autour des axes (figure 5.3)

```
#  
Graphics.moveto 0 150 ; Graphics.lineto 300 150 ;  
Graphics.moveto 2 130 ; Graphics.draw_string "absciss  
Graphics.moveto 150 0 ; Graphics.lineto 150 300 ;  
Graphics.moveto 135 280 ; draw_string_v "ordonnées" ;  
- : unit = ()
```

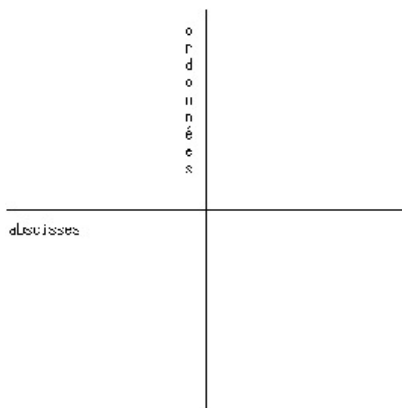


Figure 5.3 : Légende autour des axes

Si on souhaite réaliser l'affichage vertical de texte, il faut tenir compte du fait que le point courant est modifié par la fonction `draw_string_v`. Pour cela, on définit la fonction `draw_text_v` qui prend pour paramètres l'espacement entre les colonnes et la liste de mots.

```
# let draw_text_v n l =
  let f s = let (a,b) = Graphics.current_point()
    in draw_string_v s ;
    Graphics.moveto (a+n) b
  in List.iter f l ;;
val draw_text_v : int -> string list -> unit = <fun>
```

Si l'on désire effectuer d'autres transformations, comme une rotation, sur un texte, il devient nécessaire de prendre le *bitmap* de chaque lettre et d'effectuer la rotation sur ces ensembles de pixels.

Bitmaps

Un bitmap est représenté soit par une matrice de couleur (*color array array*), soit par une valeur du type abstrait¹ *image* de la bibliothèque `Graphics`. Les noms et les types des fonctions de

manipulation de bitmaps sont donnés à la figure 5.4.

fonction	type
make_image	<i>color array array</i> -> <i>image</i>
dump_image	<i>image</i> -> <i>color array array</i>
draw_image	<i>image</i> -> <i>int</i> -> <i>int</i> -> <i>unit</i>
get_image	<i>int</i> -> <i>int</i> -> <i>int</i> -> <i>int</i> -> <i>image</i>
blit_image	<i>image</i> -> <i>int</i> -> <i>int</i> -> <i>unit</i>
create_image	<i>int</i> -> <i>int</i> -> <i>image</i>

Figure 5.4 : Fonctions de manipulation de bitmaps

Les fonctions `make_image` et `dump_image` sont des fonctions de conversion entre le type *image* et le type *color array array*. La fonction `draw_image` affiche un bitmap à partir des coordonnées de son coin inférieur gauche.

De manière inverse, on peut capturer une partie rectangulaire de l'écran pour fabriquer une image par la fonction `get_image` en indiquant les coins inférieur gauche et supérieur droit de la zone à capturer. La fonction `blit_image` modifie son premier paramètre (de type *image*) en capturant la région dont le coin inférieur gauche est le point passé en paramètre. La taille de la région capturée est celle de l'image passée en argument. La fonction `create_image` permet d'initialiser des images en spécifiant leur taille pour éventuellement les utiliser avec `blit_image`.

La couleur prédéfinie `transp` est utilisée pour créer des points transparents dans une image. Cela permet d'afficher une image seulement sur une partie d'un rectangle ; les points transparents ne modifient pas l'écran initial.

Polarisation de Jussieu

Cet exemple inverse la couleur des points d'un bitmap. Pour ce faire on utilise la fonction d'inversion des couleurs présentée page ?? en l'appliquant à chaque pixel d'un bitmap.

```
# let inv_image i =  
  let inv_vec = Array.map (fun c -> inv_color c) in  
  let inv_mat = Array.map inv_vec in  
  let matrice_inversee = inv_mat (Graphics.dump_image i)  
  Graphics.make_image matrice_inversee ;;  
val inv_image : Graphics.image -> Graphics.image = <fun>
```

Soit le bitmap jussieu affiché sur la partie gauche de la figure 5.5, en utilisant la fonction `inv_image` on obtient un nouveau bitmap `< >`, comme affiché dans la partie droite de la même figure.

```
# let f_jussieu2 () = inv_image jussieu1;;  
val f_jussieu2 : unit -> Graphics.image = <fun>
```

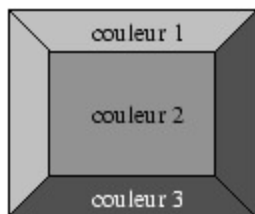


Figure 5.5 : Négation de Jussieu

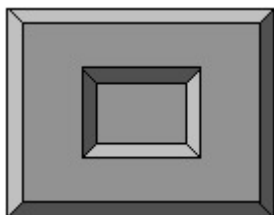
Exemple : tracé de boîtes en relief

On se propose dans cet exemple de définir quelques outils d'affichage pour le rendu du relief de boîtes. Une boîte est un objet générique qui servira dans de nombreux cas d'affichage. Une boîte est inscrite dans un rectangle caractérisé par un point d'origine, une hauteur et une largeur.

Pour donner une impression de relief à une boîte, il suffit de dessiner autour deux trapèzes de couleurs plus claires et deux autres de couleurs plus sombres.



En inversant les couleurs on peut donner l'impression que les boîtes sont pleines ou creuses.



Implantation

On ajoute aux caractéristiques d'une boîte, la largeur de sa bordure, son mode d'affichage (plein, creux ou plat), ainsi que les couleurs de ses bords et de son fond. Ces informations sont rassemblées dans un enregistrement.

```
# type relief = Top | Bot | Flat ;;  
# type box_config =  
  { x:int; y:int; w:int; h:int; bw:int; mutable r:relief  
    b1_col : Graphics.color ;  
    b2_col : Graphics.color ;  
    b_col  : Graphics.color } ;;
```

Seul le champ `r` peut être modifié. On utilise la fonction `draw_rect`, définie page ??, qui dessine un rectangle.

Par commodité, on se donne une fonction d'affichage du contour d'une boîte.

```
# let draw_box_outline bcf col =  
  Graphics.set_color col ;  
  draw_rect bcf.x bcf.y bcf.w bcf.h ;  
val draw_box_outline : box_config -> Graphics.color -> unit
```

La fonction d'affichage d'une boîte se décompose en trois parties : le dessin du premier bord, le dessin du deuxième bord et le dessin de l'intérieur de la boîte.

```
# let draw_box bcf =  
  let x1 = bcf.x and y1 = bcf.y in  
  let x2 = x1+bcf.w and y2 = y1+bcf.h in  
  let ix1 = x1+bcf.bw and ix2 = x2-bcf.bw  
  and iy1 = y1+bcf.bw and iy2 = y2-bcf.bw in  
  let border1 g =  
    Graphics.set_color g;  
    Graphics.fill_poly  
    [| (x1,y1); (ix1,iy1); (ix2,iy1); (ix2,iy2); (x2,y2); (x2,y1) |]  
  in  
  let border2 g =  
    Graphics.set_color g;  
    Graphics.fill_poly  
    [| (x1,y1); (ix1,iy1); (ix1,iy2); (ix2,iy2); (x2,y2); (x1,y2) |]  
  in  
  Graphics.set_color bcf.b_col;  
  ( match bcf.r with  
  Top ->  
    Graphics.fill_rect ix1 iy1 (ix2-ix1) (iy2-iy1) ;  
    border1 bcf.b1_col ;  
    border2 bcf.b2_col  
  | Bot ->  
    Graphics.fill_rect ix1 iy1 (ix2-ix1) (iy2-iy1) ;  
    border1 bcf.b2_col ;  
    border2 bcf.b1_col  
  | Flat ->  
    Graphics.fill_rect x1 y1 bcf.w bcf.h ) ;  
  draw_box_outline bcf Graphics.black ;;
```

```
val draw_box : box_config -> unit = <fun>
```

Le contour des boîtes est surligné en noir. L'effacement d'une boîte remplit sa zone d'affichage de la couleur du fond.

```
# let erase_box bcf =  
  Graphics.set_color bcf.b_col ;  
  Graphics.fill_rect (bcf.x+bcf.bw) (bcf.y+bcf.bw)  
    (bcf.w-(2*bcf.bw)) (bcf.h-(2*bcf.bw)) ;;  
val erase_box : box_config -> unit = <fun>
```

On définit enfin une fonction d'affichage d'une chaîne de caractères positionnée à gauche, à droite ou au centre de la boîte. On utilise le type *position* pour décrire le type de placement de la chaîne dans une boîte.

```
# type position = Left | Center | Right ;;  
type position = | Left | Center | Right  
# let draw_string_in_box pos str bcf col =  
  let (w, h) = Graphics.text_size str in  
  let ty = bcf.y + (bcf.h-h)/2 in  
  ( match pos with  
    Center -> Graphics.moveto (bcf.x + (bcf.w-w)/2) ty  
    | Right  -> let tx = bcf.x + bcf.w - w - bcf.bw - 1 in  
    Graphics.moveto tx ty  
    | Left   -> let tx = bcf.x + bcf.bw + 1 in Graphics.moveto tx ty  
    Graphics.set_color col ;  
    Graphics.draw_string str ;;  
val draw_string_in_box :  
  position -> string -> box_config -> Graphics.color -> unit
```

Exemple : tracé d'un jeu

On illustre l'utilisation des boîtes en affichant la position d'un jeu de type « morpion » comme à la figure 5.6. Pour simplifier les constructions des boîtes, on se donne des couleurs prédéfinies.

```
# let set_grey x = (Graphics.rgb x x x) ;;  
val set_grey : int -> Graphics.color = <fun>  
# let grey1= set_grey 100 and grey2= set_grey 170 and grey3= set_grey 255
```

```

val grey1 : Graphics.color = 6579300
val grey2 : Graphics.color = 11184810
val grey3 : Graphics.color = 15790320

```

On définit alors une fonction de construction d'une grille de boîtes de même taille.

```

# let rec create_grid nb_col n sep b =
  if n < 0 then []
  else
    let px = n mod nb_col and py = n / nb_col in
    let nx = b.x + sep + px*(b.w+sep)
    and ny = b.y + sep + py*(b.h+sep) in
    let bl = {b with x=nx; y=ny} in
    bl::(create_grid nb_col (n-1) sep b) ;;
val create_grid : int -> int -> int -> box_config -> box

```

Et on construit le vecteur des boîtes :

```

# let vb =
  let b = {x=0; y=0; w=20;h=20; bw=2;
    b1_col=grey1; b2_col=grey3; b_col=grey2; r=Top} in
  Array.of_list (create_grid 5 24 2 b) ;;
val vb : box_config array =
  [|{x=90; y=90; w=20; h=20; bw=2; r=Top; b1_col=6579300;
    b_col=11184810};
    {x=68; y=90; w=20; h=20; bw=2; r=Top; b1_col=6579300; b
    b_col=...};
    ...|]

```

La figure 5.6 correspond aux appels suivants :

```

# Array.iter draw_box vb ;
draw_string_in_box Center "X" vb.(5) Graphics.black ;
draw_string_in_box Center "X" vb.(8) Graphics.black ;
draw_string_in_box Center "O" vb.(12) Graphics.yellow ;
draw_string_in_box Center "O" vb.(11) Graphics.yellow ;
- : unit = ()

```

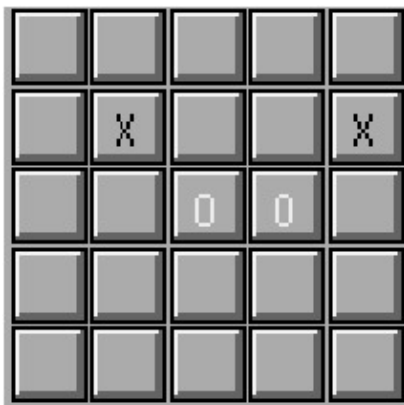


Figure 5.6 : Affichage de boîtes avec texte

Animation

L'animation d'un graphisme à l'écran reprend des techniques de dessins animés. La majeure partie du dessin ne change pas, seule la partie animée doit modifier la couleur des pixels qui la compose. Un des problèmes immédiatement rencontré provient de la vitesse d'animation. Celle-ci peut varier selon la complexité du calcul et la vitesse d'exécution du processeur. Ainsi une application graphique animée pour être portable et donner le même effet doit tenir compte de la rapidité du processeur. Pour avoir un rendu fluide il est préférable d'afficher la nouvelle position de l'objet animé, puis d'effacer l'ancienne, en tenant compte de leur intersection.

Déplacement d'un objet

Nous simplifions le problème du déplacement d'un objet en choisissant des objets de forme simple que sont les rectangles. La difficulté restante est de savoir réafficher le fond d'écran une fois l'objet déplacé.

On cherche à faire évoluer un rectangle dans un espace clos. L'objet

se déplace selon une vitesse en X et Y, quand il rencontre un bord de la fenêtre graphique, il rebondit selon son angle d'incidence. On se place dans une situation sans recouvrement entre la nouvelle et l'ancienne position de l'objet. La fonction `calc_pv` calcule, à partir d'une position (x, y), de la taille de l'objet (sx, sy) et d'une vitesse (dx, dy), la nouvelle position et la nouvelle vitesse. Cette dernière tient compte des bords de la fenêtre.

```
# let calc_pv (x,y) (sx,sy) (dx,dy) =
  let nx1 = x+dx and ny1 = y + dy
  and nx2 = x+sx+dx and ny2 = y+sy+dy
  and ndx = ref dx and ndy = ref dy
  in
    ( if (nx1 < 0) || (nx2 >= Graphics.size_x()) then ndx := -!ndx
    ( if (ny1 < 0) || (ny2 >= Graphics.size_y()) then ndy := -!ndy
    ((x+ !ndx, y+ !ndy), (!ndx, !ndy)) ;;
val calc_pv :
  int * int -> int * int -> int * int -> (int * int) * (int * int)
```

La fonction `roule_fond` déplace n fois le rectangle donné par pos et taille selon la trajectoire indiquée par sa vitesse vit en tenant compte des bords comme décrit par la fonction ci-dessus. La trace du déplacement que l'on obtient sur la figure 5.7 est obtenue par inversion du bitmap correspondant au rectangle déplacé.

```
# let roule_fond pos taille vit n =
  let (x, y) = pos and (sx,sy) = taille in
  let mem = ref (Graphics.get_image x y sx sy) in
  let rec roule_aux x y vit n =
    if n = 0 then Graphics.moveto x y
    else
      let ((nx,ny),n_vit) = calc_pv (x,y) (sx,sy) vit
      and old_mem = !mem in
      mem := Graphics.get_image nx ny sx sy ;
      Graphics.set_color Graphics.blue;
      Graphics.fill_rect nx ny sx sy;
      Graphics.draw_image (inv_image old_mem) x y;
      roule_aux nx ny n_vit (n-1)
  in roule_aux x y vit n ;;
```

```
val roule_fond : int * int -> int * int -> int * int ->
```

Le code suivant correspond aux dessins de la figure 5.7. Le premier est obtenu sur fond uniformément rouge, le second, en déplaçant le rectangle sur l'image de Jussieu.

```
# let anim_rect () =  
  Graphics.moveto 105 120 ;  
  Graphics.set_color Graphics.white;  
  Graphics.draw_string "Début" ;  
  roule_fond (140,120) (8,8) (8,4) 150 ;  
  let (x,y) = Graphics.current_point() in  
  Graphics.moveto (x+13) y ;  
  Graphics.set_color Graphics.white;  
  Graphics.draw_string "Fin" ;;  
val anim_rect : unit -> unit = <fun>  
# anim_rect();;  
- : unit = ()
```



Figure 5.7 : Code du déplacement d'un objet

Le problème a été simplifié dans la mesure où il n'y a pas d'intersection entre deux positions successives de l'objet déplacé. Si tel n'est pas le cas, il faut écrire une fonction de calcul de cette intersection qui est plus ou moins compliquée selon la forme de l'objet. Dans le cas présent d'un carré, l'intersection de deux carrés donne un rectangle. C'est celui-ci qu'il faut alors effacer.

Événements

La gestion des événements produits dans la fenêtre graphique permet de construire une interaction conviviale entre l'utilisateur et le programme. Les événements traités par Graphics sont l'appui

d'une touche au clavier, le clic de la souris et le mouvement de celle-ci.

Le style de programmation change ainsi que l'organisation du programme. Celui-ci devient une boucle sans fin d'attente d'événements. À chaque nouveau déclenchement d'un événement, celui-ci est traité, puis le programme revient dans la boucle d'attente sauf lorsqu'un événement est prévu pour provoquer la sortie du programme.

Types et fonctions sur les événements

La fonction principale d'attente d'événements est `wait_next_event` de type *event list* -> *status*.

Les différents événements sont donnés par le type somme *event*.

```
type event = Button_down | Button_up | Key_pressed | Mou
```

Les quatre premières valeurs correspondent, respectivement, à l'appui ou au relâchement d'un bouton de souris, au déplacement de la souris et à l'appui d'une touche. L'attente est bloquante, sauf si le constructeur `Poll` est ajouté à la liste des événements. Cette fonction retourne une valeur de type *status* :

```
type status =  
  { mouse_x : int;  
    mouse_y : int;  
    button : bool;  
    keypressed : bool;  
    key : char};;
```

C'est un enregistrement contenant la position de la souris, un booléen indiquant si un bouton est pressé, un autre booléen pour le clavier et un caractère correspondant au caractère de la touche utilisée. Les fonctions suivantes exploitent les données de l'enregistrement des événements :

- `mouse_pos : unit -> int * int` : retourne la position de la souris

par rapport à la fenêtre, si la souris est ailleurs, les coordonnées sont en dehors des bornes de la fenêtre.

- `button_down : unit -> bool` : indique la pression sur un bouton de la souris.
- `read_key : unit -> char` : renvoie un caractère tapé au clavier; l'attente est bloquante.
- `key_pressed : unit -> bool` : indique si une touche du clavier est pressée; l'attente est non bloquante.

La gestion d'événements proposée par Graphics est vraiment minimale pour la construction d'interfaces interactives. Néanmoins, le code est portable sur différents systèmes graphiques comme Windows, MacOS ou X-Windows. C'est d'ailleurs pour cela que cette bibliothèque ne tient pas compte des différents boutons de la souris. En effet les Mac n'en possèdent qu'un seul. Les autres événements : exposition et changement de taille de la fenêtre ne sont pas accessibles et sont laissés à la charge de la bibliothèque.

Squelette de programme

Tous les programmes d'interface utilisateur comprennent une boucle potentiellement infinie d'attente d'action de l'utilisateur. Une fois celle-ci transmise, le programme effectue le travail lié à cette action. La fonction suivante possède cinq paramètres fonctionnels. Les deux premiers servent à démarrer et à fermer l'application. Les deux suivants sont les fonctions de traitement des événements clavier et souris. Le dernier permet la gestion des exceptions lors des différents calculs de l'application. On suppose que les événements associés à la sortie de l'application déclenchent l'exception `Fin`.

```
# exception Fin;;
exception Fin
# let squel f_init f_end f_key f_mouse f_except =
  f_init () ;
  try
    while true do
      try
        let s = Graphics.wait_next_event
```



```

[Graphics.Button_down; Graphics.Key_pressed]
in if s.Graphics.keypressed then f_key s.Graphics.key
else if s.Graphics.button
then f_mouse s.Graphics.mouse_x s.Graphics.mouse_y
with
Fin -> raise Fin
| e -> f_except e
done
with
Fin -> f_end () ;;
val squel :
  (unit -> 'a) ->
  (unit -> unit) ->
  (char -> unit) -> (int -> int -> unit) -> (exn -> unit)

```

Le squelette est utilisé pour programmer une mini-machine à écrire. L'appui d'une touche affiche le caractère tapé. Le clic d'une souris change le point courant. Le caractère '\$' fait sortir du programme. La seule difficulté de ce programme est le saut de ligne. On suppose, pour simplifier, que la hauteur des caractères n'excède pas douze pixels.

```

# let next_line () =
  let (x,y) = Graphics.current_point()
  in if y>12 then Graphics.moveto 0 (y-12)
  else Graphics.moveto 0 y ;;
val next_line : unit -> unit = <fun>
# let trait_char c = match c with
  '$' -> raise Fin
| '\n' -> next_line ()
| '\r' -> next_line ()
| _ -> Graphics.draw_char c ;;
val trait_char : char -> unit = <fun>
# let go () = squel
  (fun () -> Graphics.clear_graph () ;
  Graphics.moveto 0 (Graphics.size_y() -12) )
  (fun () -> Graphics.clear_graph())
  trait_char
  (fun x y -> Graphics.moveto x y)
  (fun e -> ()) ;;

```

```
val go : unit -> unit = <fun>
```

Ce programme ne gère pas l'effacement d'un caractère par l'appui de la touche DEL.

Exemple : le télécran

Le télécran est ce petit jeu de dessin pour la coordination des mouvements. Un point apparaît sur une ardoise. Ce point peut se déplacer en X et en Y en utilisant deux boutons de contrôle de ces axes, sans jamais relever le crayon. On cherche à simuler ce comportement pour illustrer l'interaction entre un programme et un utilisateur. Pour cela on reprendra le squelette décrit précédemment. On utilisera certaines touches du clavier pour indiquer un mouvement sur un des axes.

On définit tout d'abord le type *etat* qui est un enregistrement décrivant la taille de l'ardoise en nombre de positions en X et en Y, la position courante du point, le facteur d'échelle pour la visualisation, la couleur du trait, la couleur du fond et la couleur pour repérer le point courant.

```
# type etat = {maxx:int; maxy:int; mutable x : int; mutable y : int;
  scale:int;
  bc : Graphics.color;
  fc: Graphics.color; pc : Graphics.color} ;;
```

La fonction `draw_point` affiche un point en lui donnant comme paramètres les coordonnées, le facteur d'échelle et la couleur.

```
# let draw_point x y s c=
  Graphics.set_color c;
  Graphics.fill_rect (s*x) (s*y) s s ;;
val draw_point : int -> int -> int -> Graphics.color ->
```

Toutes les fonctions d'initialisation, de gestion de l'interaction et de sortie du programme recevront un paramètre correspondant à l'état. Les quatre premières fonctions sont définies ci-après :

```

# let t_init e () =
  Graphics.open_graph (":0 " ^ (string_of_int (e.scale*e.e.
    "x" ^ (string_of_int (e.scale*e.maxy))) ) ;
  Graphics.set_color e.bc ;
  Graphics.fill_rect 0 0 (e.scale*e.maxx+1) (e.scale*e.ma
    draw_point e.x e.y e.scale e.pc ;;
val t_init : etat -> unit -> unit = <fun>
# let t_end e () =
  Graphics.close_graph() ;
  print_string "A bientôt..."; print_newline() ;;
val t_end : 'a -> unit -> unit = <fun>
# let t_mouse e x y = () ;;
val t_mouse : 'a -> 'b -> 'c -> unit = <fun>
# let t_except e ex = () ;;
val t_except : 'a -> 'b -> unit = <fun>

```

La fonction `t_init` ouvre la fenêtre graphique et affiche le point courant, `t_end` ferme cette fenêtre et affiche un message, `t_mouse` et `t_except` ne font rien. Le programme ne gèrera pas les événements souris, ni les exceptions qui risquent de se déclencher à l'exécution. La fonction importante est celle de la gestion du clavier `t_key` :

```

# let t_key e c =
  draw_point e.x e.y e.scale e.fc;
  (match c with
  | '8' -> if e.y < e.maxy then e.y <- e.y + 1;
  | '2' -> if e.y > 0 then e.y <- e.y - 1
  | '4' -> if e.x > 0 then e.x <- e.x - 1
  | '6' -> if e.x < e.maxx then e.x <- e.x + 1
  | 'c' -> Graphics.set_color e.bc;
  Graphics.fill_rect 0 0 (e.scale*e.maxx+1) (e.scale*e.ma
  Graphics.clear_graph()
  | 'f' -> raise Fin
  | _ -> ());
  draw_point e.x e.y e.scale e.pc;;
val t_key : etat -> char -> unit = <fun>

```

Elle affiche le point courant de la couleur du trait, et selon le caractère passé modifie, si possible, les coordonnées du point courant (caractères : '2', '4', '6', '8'), efface l'écran (caractère : 'c') ou

déclenche l'exception Fin (caractère : 'f'), puis affiche le nouveau point courant. Les caractères autres que ceux mentionnés sont ignorés. Le choix des caractères de déplacement du curseur provient de la disposition du pavé numérique : les touches choisies correspondent aux chiffres indiqués et aux flèches de direction. Il est utile alors d'activer le pavé numérique pour l'ergonomie du programme.

On définit enfin un état et on applique la fonction squelette de la manière suivante :

```
# let etel = {maxx=120;maxy=120; x= 60; y= 60;
  scale=4; bc=Graphics.rgb 130 130 130;
  fc=Graphics.black; pc = Graphics.red};;
val etel : etat =
  {maxx=120; maxy=120; x=60; y=60; scale=4; bc=8553090; f
# let ardoise () =
  squel (t_init etel) (t_end etel) (t_key etel)
  (t_mouse etel) (t_except etel);;
val ardoise : unit -> unit = <fun>
```

L'appel à ardoise affiche la fenêtre graphique puis attend une interaction clavier. La figure 5.8 montre un dessin réalisé par ce programme.

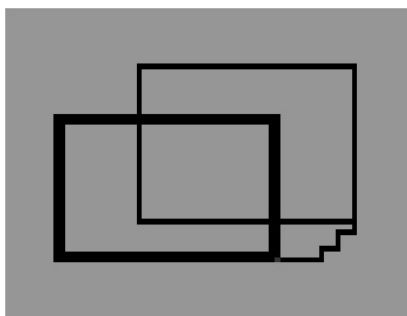


Figure 5.8 : Le télécran

Calculatrice graphique

Nous reprenons l'exemple de la calculatrice décrite dans le chapitre précédent sur la programmation impérative (voir page ??). Nous la dotons d'une interface graphique la rendant plus facile à être utilisée comme une calculatrice de bureau.

L'interface graphique matérialisera l'ensemble des touches (chiffres et fonctions) et une zone de visualisation des résultats. Une touche pourra être activée soit via l'interface graphique (et la souris) soit directement au clavier. La figure 5.9 montre l'interface que l'on désire construire.



Figure 5.9 : Calculatrice graphique

On réutilise les fonctions de tracé de boîtes décrites à la page ?? . On définit le type suivant :

```
# type etat_calc =  
  { e : etat; t : (box_config * touche * string ) list; v
```

Il permet de conserver l'état de la calculatrice, la liste des boîtes

correspondant aux touches et la boîte de visualisation. On désire construire une calculatrice graphique facilement modifiable. On paramètre alors la construction de l'interface par une liste d'associations :

```
# let descr_calc =
[ (Chiffre 0,"0"); (Chiffre 1,"1"); (Chiffre 2,"2");
  (Chiffre 3,"3"); (Chiffre 4,"4"); (Chiffre 5,"5");
  (Chiffre 6,"6"); (Chiffre 7,"7"); (Chiffre 8,"8");
  (Chiffre 9,"9"); (MemoireOut,"RCL"); (Par, "/");
  (Off,"AC"); (MemoireIn, "STO"); (Clear,"CE/C")
] ;;
```

Génération des boîtes des touches

À partir de cette description on construit la liste des boîtes des touches. La fonction `gen_boxes` prend pour paramètres une description (`descr`), le nombre de colonnes (`n`), la séparation entre boîtes (`wsep`), la séparation entre le texte et les bords d'une boîte (`wsepint`) et la taille des bords (`wbord`). Cette fonction retourne la liste des boîtes des touches ainsi que la boîte de visualisation. Pour calculer ces placements, on définit les fonctions auxiliaires `max_xy` de calcul des tailles maximales d'une liste de couples d'entiers et `max_lbox` de calcul des positions maximales d'une liste de boîtes.

```
# let gen_xy vals comp o =
  List.fold_left (fun a (x,y) -> comp (fst a) x,comp (snd a) y) vals
val gen_xy : ('a * 'a) list -> ('b -> 'a -> 'b) -> 'b * 'a
# let max_xy vals = gen_xy vals max (min_int,min_int);;
val max_xy : (int * int) list -> int * int = <fun>
# let max_boxl l =
  let bmax (mx,my) b = max mx b.x, max my b.y
  in List.fold_left bmax (min_int,min_int) l ;;
val max_boxl : box_config list -> int * int = <fun>
```

Voici la fonction principale `gen_boxes` de création de l'interface.

```
# let gen_boxes descr n wsep wsepint wbord =
  let l_l = List.length descr in
```

```

let nb_lig = if l_1 mod n = 0 then l_1 / n else l_1 / n + 1
let ls = List.map (fun (x,y) -> Graphics.text_size y) c
let sx,sy = max_xy ls in
let sx,sy= sx+wsepint ,sy+wsepint in
let r = ref [] in
for i=0 to l_1-1 do
let px = i mod n and py = i / n in
let b = { x = wsep * (px+1) + (sx+2*wbord) * px ;
y = wsep * (py+1) + (sy+2*wbord) * py ;
w = sx; h = sy ; bw = wbord;
r=Top;
b1_col = grey1; b2_col = grey3; b_col =grey2}
in r:= b::!r
done;
let mpx,mpy = max_boxl !r in
let upx,upy = mpx+sx+wbord+wsep,mpy+sy+wbord+wsep in
let (wa,ha) = Graphics.text_size " 0" in
let v = { x=(upx-(wa+wsepint +wbord))/2 ; y= upy+ wsep;
w=wa+wsepint; h = ha +wsepint; bw = wbord *2; r=Flat ;
b1_col = grey1; b2_col = grey3; b_col =Graphics.black}
in
upx, (upy+wsep+ha+wsepint+wsep+2*wbord),v,
List.map2 (fun b (x,y) -> b,x,y ) (List.rev !r) descr;;
val gen_boxes :
('a * string) list ->
int ->
int ->
int -> int -> int * int * box_config * (box_config * 'a)
<fun>

```

Interaction

Comme l'on désire aussi reprendre le squelette proposé page ?? pour l'interaction, on définit les fonctions de gestion du clavier et de la souris qui s'intègrent à ce squelette. La fonction de gestion du clavier est fort simple. Elle passe la traduction du caractère en valeur de type *touche* à la fonction transition de la calculatrice, puis affiche le texte de l'état de la calculatrice.

```
# let f_clavier ec c =
  transition ec.e (traduction c);
  draw_string_in_box Right (string_of_int ec.e.vaf) ec.v
val f_clavier : etat_calc -> char -> unit = <fun>
```

La gestion de la souris est un peu plus complexe. Elle nécessite de vérifier que la position du clic souris est bien dans une des boîtes des touches. Pour cela on définit tout d'abord la fonction auxiliaire `mem` qui vérifie l'appartenance d'une position à un rectangle.

```
# let mem (x,y) (x0,y0,w,h) =
  (x >= x0) && (x < x0+w) && (y >= y0) && (y < y0+h);;
val mem : int * int -> int * int * int * int -> bool = <fun>
# let f_souris ec x y =
  try
    let b,t,s =
      List.find (fun (b,_,_) ->
        mem (x,y) (b.x+b.bw,b.y+b.bw,b.w,b.h)) ec.t
    in
      transition ec.e t;
      draw_string_in_box Right (string_of_int ec.e.vaf) ec.v
  with Not_found -> ();;
val f_souris : etat_calc -> int -> int -> unit = <fun>
```

La fonction `f_souris` cherche si la position de la souris lors du clic est bien dans une des boîtes correspondant à une touche, si oui elle passe la touche correspondante à la fonction de transition puis affiche le résultat, sinon elle ne fait rien.

La fonction `f_exc` gère les exceptions pouvant se déclencher lors de l'exécution de ce programme.

```
# let f_exc ec ex =
  match ex with
  Division_by_zero ->
    transition ec.e Clear;
    draw_string_in_box Right "Div 0" ec.v (Graphics.red)
  | Touche_non_valide -> ()
  | Touche_off -> raise Fin
  | _ -> raise ex;;
val f_exc : etat_calc -> exn -> unit = <fun>
```


Si une division par zéro survient, elle remet dans l'état initial la calculatrice et affiche un message d'erreur à l'écran de la calculatrice. Une touche non valide est simplement ignorée. Enfin l'exception `Touche_off` déclenche l'exception `Fin` de sortie de la boucle du squelette.

Initialisation et sortie

L'initialisation de la calculatrice nécessite de calculer la taille de la fenêtre. La fonction suivante engendre les informations graphiques des boîtes à partir d'une association touche-texte et retourne la taille de la fenêtre principale.

```
# let create_e t =
  Graphics.close_graph ();
  Graphics.open_graph " 10x10";
  let mx,my,v,lb = gen_boxes t 4 4 5 2 in
  let s = {dce=0; dta = false; doa = Egal; vaf = 0; mem =
    mx,my,{e=s; t=lb;v=v}};;
val create_e : (touche * string) list -> int * int * etat
```

La fonction d'initialisation utilise le résultat de la fonction précédente.

```
# let f_init mx my ec () =
  Graphics.close_graph();
  Graphics.open_graph (":0 "^(string_of_int mx)^"x"^(string_of_int my));
  Graphics.set_color grey2;
  Graphics.fill_rect 0 0 (mx+1) (my+1);
  List.iter (fun (b,_,_) -> draw_box b) ec.t;
  List.iter
    (fun (b,_,s) -> draw_string_in_box Center s b Graphics.set_color)
    ec.v;
  draw_string_in_box Right "hello" ec.v (Graphics.white);
val f_init : int -> int -> etat_calc -> unit -> unit = <
```

Enfin la fonction de sortie ferme la fenêtre graphique.

```
# let f_fin e () = Graphics.close_graph();;
val f_fin : 'a -> unit -> unit = <fun>
```

La fonction `go`, paramétrée par une description lance la boucle d'interaction.

```
# let go descr =  
  let mx,my,e = create_e descr in  
  squel (f_init mx my e) (f_fin e) (f_clavier e) (f_souri  
val go : (touche * string) list -> unit = <fun>
```

L'appel `go descr_calc` correspond à la figure 5.9.

Exercices

Coordonnées polaires

Les coordonnées de la bibliothèque Graphics sont cartésiennes. Un segment y est représenté par son point de départ (x_0, y_0) et son point d'arrivée (x_1, y_2) Il peut être utile d'utiliser des coordonnées polaires. Un segment est alors décrit par son point de départ (x_0, y_0) la taille du segment (r) et son angle (a) Le rapport entre coordonnées cartésiennes et coordonnées polaires est définie par les équations :

$$\begin{matrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{matrix}$$

Le type suivant définit les coordonnées polaires d'un segment :

```
# type seg_pol = {x:float; y:float; r:float; a:float};;  
type seg_pol = { x: float; y: float; r: float; a: float
```

1. Écrire la fonction `to_cart` de transformation de coordonnées polaires en coordonnées cartésiennes.

```
# let to_cart p =  
  (p.x, p.y), (p.x +. p.r *. (cos p.a), p.y +. p.r *.  
val to_cart : seg_pol -> (float * float) * (float *
```

2. Écrire la fonction `draw_seg` qui affiche un segment défini en coordonnées polaires dans le repère de Graphics.

```
# let draw_seg p =  
  let (x1,y1), (x2,y2) = to_cart p in  
  Graphics.moveto (int_of_float x1) (int_of_float y1)  
  Graphics.lineto (int_of_float x2) (int_of_float y2)  
val draw_seg : seg_pol -> unit = <fun>
```

3. Un des intérêts des coordonnées polaires est de pouvoir facilement appliquer des transformations sur un segment. Une translation ne modifiera que le point de départ, une rotation sera uniquement effectuée sur le champ angle, de même pour un changement d'échelle sur le champ longueur. De façon générale, on peut représenter une transformation comme un triplet de flottants : le premier représente la translation (on ne considère, ici, que des translations sur la droite du segment), le second la rotation et le troisième le facteur d'échelle. Définir la fonction `app_trans` qui prend un segment en coordonnées polaires et un triplet de transformations et retourne le nouveau segment.

```
# let app_trans seg ( da, dr, dxy ) =  
  let _, (x1,y1) = to_cart {seg with r = seg.r *. dxy  
    {x=x1; y=y1; a=seg.a +. da; r=seg.r *. dr} ;;  
val app_trans : seg_pol -> float * float * float ->
```

4. On peut construire des dessins récursifs par itération de transformations. Écrire la fonction `dessin_r` qui prend en arguments un segment `s`, un nombre d'itérations `n`, une liste de transformations `l` et affiche tous les segments résultant des transformations sur `s` itérées jusqu'à `n`.

```
# let rec dessin_r s n l =
```

```

if n = 0 then ()
else
begin
  draw_seg s;
  List.iter (fun t -> dessin_r (app_trans s t) (n-1))
end ;;
val dessin_r : seg_pol -> int -> (float * float * float)

```

5. **Vérifier** que le programme suivant produit bien les images de la figure 5.10.

```

let pi = 3.1415927 ;;
let s = {x=100.; y= 0.; a= pi /. 2.; r = 100.} ;;
dessin_r s 6 [ (-.pi/.2.), 0.6, 1.; (pi/.2.), 0.6, 1.0
Graphics.clear_graph();;
dessin_r s 6 [(-.pi /. 6.), 0.6, 0.766;
  (-.pi /. 4.), 0.55, 0.333;
  (pi /. 3.), 0.4, 0.5 ] ;;

# Graphics.close_graph();;
- : unit = ()
# Graphics.open_graph ":0 200x200";;
- : unit = ()
# let pi = 3.1415927 ;;
val pi : float = 3.1415927
# let s = {x=100.; y= 0.; a= pi /. 2.; r = 100.} ;;
val s : seg_pol = {x=100; y=0; r=100; a=1.57079635}
# dessin_r s 6 [ (-.pi/.2.), 0.6, 1.; (pi/.2.), 0.6, 1
- : unit = ()
# Graphics.clear_graph();;
- : unit = ()
# dessin_r s 6 [(-.pi /. 6.), 0.6, 0.766;
  (-.pi /. 4.), 0.55, 0.333;
  (pi /. 3.), 0.4, 0.5 ] ;;
- : unit = ()

```



Figure 5.10 : dessins récursifs

Éditeur de bitmaps

On cherche à un écrire un petit éditeur de bitmap (à la manière de la commande `bitmap` de X-window). Pour cela on représente un bitmap par ses dimensions (largeur et hauteur), la taille des pixels, et un tableau à 2 dimensions de booléens.

1. Définir un type `etat_bitmap` décrivant l'information nécessaire pour conserver les valeurs des pixels, la taille du bitmap et les couleurs d'affichage et d'effacement.

```
# type etat_bitmap =  
  {w : int; h : int; fg : Graphics.color; bg : Graph  
  pix : bool array array; s : int} ;;  
type etat_bitmap =  
  { w: int;  
    h: int;  
    fg: Graphics.color;  
    bg: Graphics.color;  
    pix: bool array array;  
    s: int }
```

2. Écrire les fonctions de création d'un bitmap (`create_bitmap`) et d'affichage d'un bitmap (`draw_bitmap`) .

```
# let create_bitmap x y f g t =
```

```

let r = Array.make_matrix x y false in
{ w = x; h = y; fg = f; bg = g; pix = r; s = t } ;
val create_bitmap :
  int -> int -> Graphics.color -> Graphics.color ->
  <fun>

# let draw_pix i j s c =
  Graphics.set_color c;
  Graphics.fill_rect (i*s+1) (j*s+1) (s-1) (s-1) ;;
val draw_pix : int -> int -> int -> Graphics.color

# let draw_bitmap b =
  for i=0 to b.w-1 do
    for j=0 to b.h-1 do
      draw_pix i j b.s (if b.pix.(i).(j) then b.fg else
      done
    done ;;
val draw_bitmap : etat_bitmap -> unit = <fun>

```

3. Écrire les fonctions [read_bitmap](#) et [write_bitmap](#) qui respectivement lit et écrit un bitmap dans un fichier passé en paramètre en suivant le format ASCII de X-window. Si le fichier n'existe pas, alors la fonction de lecture crée un nouveau bitmap en utilisant la fonction `create_bitmap`. Un pixel allumé est représenté par le caractère #, l'absence d'un pixel par le caractère -. Chaque ligne de caractères représente une ligne du bitmap. On pourra tester le programme en utilisant les fonctions `atobm` et `bmtoa` de X-window qui effectueront les conversions entre ce format ASCII et le format des bitmaps créés par la commande `bitmap`. Voici un exemple.

```

<font size="1">
#####-----#####-----##
#####-----###-----
###-----###-----###-----
##-----###-----##-----##

```



```

let b = create_bitmap w h Graphics.black Graphics.
for j = 0 to h - 1 do
  for i = 0 to w - 1 do
    b.pix.(i).(j) <- ( r.(j).[i] = '#' )
  done
done;
b ;;

val read_bitmap : string -> etat_bitmap = <fun>

# let save_bitmap filename b =
  let oc = open_out filename in
  let f x = output_char oc (if x then '#' else '-')
  Array.iter (fun x -> (Array.iter f x)); output_char
  close_out oc ;;
val save_bitmap : string -> etat_bitmap -> unit = <

```

4. On reprend le [squelette](#) de boucle d'interaction de la page ?? pour construire l'interface graphique de l'éditeur. L'interface homme-machine est fort simple. Le bitmap est affiché en permanence dans la fenêtre graphique. Un clic souris sur une des cases du bitmap inverse sa valeur. Ce changement est répercuté à l'écran. L'appui sur la touche 'S' sauve le bitmap dans le fichier. L'appui sur la touche 'Q' fait sortir du programme.

- Écrire la fonction [start](#) de type *etat_bitmap -> unit -> unit* qui ouvre la fenêtre graphique et affiche le bitmap passé en paramètre.

```

# exception Fin ;;
exception Fin
# let sqel f_init f_end f_key f_mouse f_except
  f_init ();
  try
    while true do
      try
        let s = Graphics.wait_next_event
        [Graphics.Button_down; Graphics.Key_pressed] f
        if s.Graphics.keypressed

```



```

then f_key s.Graphics.key
else if s.Graphics.button
then f_mouse s.Graphics.mouse_x s.Graphics.mouse_y
with
  Fin -> raise Fin
  | e -> f_except e
done
with
  Fin -> f_end () ;;
val squel :
  (unit -> 'a) ->
  (unit -> unit) ->
  (char -> unit) -> (int -> int -> unit) -> (exn -> unit) -> 'a

# let start b () =
  let ow = 1+b.w*b.s and oh = 1+b.h*b.s in
  Graphics.open_graph (":0 " ^ (string_of_int ow))
  Graphics.set_color (Graphics.rgb 150 150 150)
  Graphics.fill_rect 0 0 ow oh ;
  draw_bitmap b ;;
val start : etat_bitmap -> unit -> unit = <fun>

```

- Écrire la fonction **stop** qui ferme la fenêtre graphique et sort du programme.

```

# let stop () = Graphics.close_graph() ; exit 0
val stop : unit -> 'a = <fun>

```

- Écrire la fonction **mouse** de type *etat_bitmap -> int -> int -> unit* qui modifie l'état du pixel correspondant au clic souris et affiche ce changement.

```

# let mouse b x y =
  let i,j = (x / b.s), (y/b.s) in
  if ( i < b.w ) && ( j < b.h ) then
    begin
      b.pix.(i).(j) <- not b.pix.(i).(j) ;
      draw_pix i j b.s (if b.pix.(i).(j) then b.fg else b.bg)
    end

```

```

    end ;;
    val mouse : etat_bitmap -> int -> int -> unit =

```

- Écrire la fonction **key** de type *string -> etat_bitmap -> char -> unit* qui prend en paramètre un nom de fichier, un bitmap et le caractère de la touche appuyée et effectue les actions associées : sauvegarde dans un fichier pour la touche 'S' et déclenchement de l'exception Fin pour la touche 'Q'.

```

# let key filename b c =
    match c with
    | 'q' | 'Q' -> raise Fin
    | 's' | 'S' -> save_bitmap filename b
    | _ -> () ;;
    val key : string -> etat_bitmap -> char -> unit

```

9. Écrire une fonction **go** qui prend en paramètre un nom de fichier, charge le bitmap puis l'affiche et lance la boucle d'interaction.

```

# let go name =
    let b = try
        read_bitmap name
    with
    _ -> create_bitmap 10 10 Graphics.black Graphics.w
    in squel (start b) stop (key name b) (mouse b) (fun
    val go : string -> unit = <fun>

```

Ver de Terre

Le ver de terre est un petit organisme, longiforme, d'une certaine taille qui va croître soit avec le temps soit en ramassant des objets dans un monde. Le ver de terre est toujours en mouvement selon

une direction. Ses actions propres ou dirigées par un joueur permettent uniquement un changement de direction. Le ver de terre disparaît s'il touche un bord du monde ou s'il passe sur une partie de son corps. On le représente le plus souvent par un vecteur de coordonnées avec deux indices principaux : sa tête et sa queue. Un mouvement sera donc le calcul des nouvelles coordonnées de sa tête, et son affichage, et l'effacement de sa queue. Une croissance ne modifiera que sa tête sans toucher à la queue du ver de terre.

1. Écrire le ou les **types** Objective CAML pour représenter un ver de terre et le monde où il évolue. On peut représenter un ver de terre par une file d'attente de ses coordonnées.

```
# type cell = Vide | Pleine ;;
type cell = | Vide | Pleine

# type monde = { l : int; h : int; cases : cell array
type monde = { l: int; h: int; cases: cell array ar

# type vdt = { mutable tete : int; mutable queue :
mutable vx : int; mutable vy : int;
pos : (int * int) array } ;;
type vdt =
{ mutable tete: int;
mutable queue: int;
mutable taille: int;
mutable vx: int;
mutable vy: int;
pos: (int * int) array }

# type jeu = { m : monde; v : vdt; t_cell : int;
fg : Graphics.color; bg : Graphics.color } ;;
type jeu =
{ m: monde;
v: vdt;
t_cell: int;
fg: Graphics.color;
bg: Graphics.color }
```

2. Écrire les fonctions d'**initialisation** et d'**affichage** d'un ver de terre dans un monde.

```
# let init_monde larg haut =
  { l = larg; h = haut; cases = Array.create_matrix
val init_monde : int -> int -> monde = <fun>

# let init_vdt t t_max larg =
  if t > larg then failwith "init_vdt"
  else
    begin
      let v = { tete = t-1; queue = 0; taille = t; vx =
pos = Array.create t_max (0,0) } in
      let y = larg / 2
      and rx = ref (larg/2 - t/2) in
      for i=0 to t-1 do v.pos.(i) <- (!rx,y) ; incr rx d
      v
    end ;;
val init_vdt : int -> int -> int -> vdt = <fun>

# let init_jeu t t_max larg haut tc c1 c2 =
  let j = { m = init_monde larg haut; v = init_vdt t
t_cell = tc; fg = c1; bg = c2 } in
  for i=j.v.tete to j.v.queue do
    let (x,y) = j.v.pos.(i) in
    j.m.cases.(x).(y) <- Pleine
  done ;
  j ;;
val init_jeu :
  int -> int -> int -> int -> int -> Graphics.color
  <fun>

# let affiche_case x y t c =
  Graphics.set_color c;
  Graphics.fill_rect (x*t) (y*t) t t ;;
val affiche_case : int -> int -> int -> Graphics.co

# let affiche_monde jeu =
```

```

let m = jeu.m in
for i=0 to m.l-1 do
for j=0 to m.h-1 do
let col = if m.cases.(i).(j) = Pleine then jeu.fg
affiche_case i j jeu.t_cell col
done
done ;;
val affiche_monde : jeu -> unit = <fun>

```

3. Modifier la fonction **squel** de squelette du programme qui à chaque tour dans la boucle d'interaction effectue une action, paramétrée par une fonction. La gestion de l'événement clavier ne doit pas être bloquante.

```

(***** interaction *****)

# let tempo ti =
  for i = 0 to ti do ignore (i * ti * ti) done ;;
val tempo : int -> unit = <fun>

# exception Fin;;
exception Fin
# let squel f_init f_end f_key f_mouse f_except f_r
  f_init ();
  try
  while true do
  try
  tempo 200000 ;
  if Graphics.key_pressed() then f_key (Graphics.rea
  f_run ()
  with
  Fin -> raise Fin
  | e -> f_except e
  done
  with
  Fin -> f_end () ;;
val squel :
  (unit -> 'a) ->

```

```
(unit -> unit) ->
(char -> unit) -> 'b -> (exn -> 'c) -> (unit -> 'c
```

4. Écrire la fonction **run** qui fait avancer le ver de terre dans le jeu. Cette fonction déclenche les exceptions Gagne (si le ver a atteint une certaine taille) et Perdu s'il rencontre une case pleine ou un bord du monde.

```
# exception Perdu;;
exception Perdu
# exception Gagne;;
exception Gagne

# let run jeu temps itemps () =
  incr temps;
  let v = jeu.v in
  let t = Array.length v.pos in
  let ox,oy = v.pos.(v.tete) in
  let nx,ny = ox+v.vx,oy+v.vy in
  if (nx < 0 ) || (nx >= jeu.m.l) || (ny < 0) || (ny
  then raise Perdu
  else if jeu.m.cases.(nx).(ny) = Pleine then raise
  else if v.tete = v.queue then raise Gagne
  else
  begin
    let ntete = (v.tete + 1) mod t in
    v.tete <- ntete ;
    v.pos.(v.tete) <- (nx,ny);
    jeu.m.cases.(nx).(ny) <- Pleine ;
    affiche_case nx ny jeu.t_cell jeu.fg ;
    if (!temps mod !itemps < (!itemps - 2)) then
    begin
      let qx,qy = v.pos.(v.queue) in
      jeu.m.cases.(qx).(qy) <- Vide ;
      affiche_case qx qy jeu.t_cell jeu.bg ;
      v.queue <- (v.queue + 1) mod t
    end
  end ;;
```

```
val run : jeu -> int ref -> int ref -> unit -> unit
```

5. Écrire la fonction d'[interaction](#) clavier qui modifie la direction du ver de terre.

```
# let key lact jeu c =  
  match c with  
  | 'q' | 'Q' -> raise Fin  
  | 'p' | 'P' -> ignore (Graphics.read_key ())  
  | '2' | '4' | '6' | '8' ->  
    let dx,dy = List.assoc c lact in  
    jeu.v.vx <- dx ;  
    jeu.v.vy <- dy  
  | _ -> () ;;  
val key : (char * (int * int)) list -> jeu -> char
```

6. Écrire les autres [fonctions utilitaires](#) de gestion de l'interaction et les passer au nouveau squelette de programme.

```
# let start jeu () =  
  let ow = jeu.t_cell * jeu.m.l and oh = jeu.t_cell  
  let size = (string_of_int ow) ^ "x" ^ (string_of_int oh)  
  Graphics.open_graph (":0 " ^ size) ;  
  Graphics.set_color (Graphics.rgb 150 150 150);  
  Graphics.fill_rect 0 0 ow oh;  
  affiche_monde jeu;  
  ignore (Graphics.read_key()) ;;  
val start : jeu -> unit -> unit = <fun>  
  
# let stop jeu () =  
  ignore (Graphics.read_key());  
  Graphics.close_graph() ;;  
val stop : 'a -> unit -> unit = <fun>  
  
# let mouse x y = () ;;  
val mouse : 'a -> 'b -> unit = <fun>
```

```
# let except e = match e with
  Perdu -> print_endline "PERDU"; raise Fin
  | Gagne -> print_endline "GAGNE"; raise Fin
  | e -> raise e ;;
val except : exn -> 'a = <fun>
```

7. Écrire la **fonction principale** de lancement de l'application.

```
# let la = [ ('2', (0, -1)) ; ('4', (-1, 0)) ; ('6', (1,
val la : (char * (int * int)) list =
  ['2', (0, -1); '4', (-1, 0); '6', (1, 0); '8', (0,

# let go larg haut tc =
  let col = Graphics.rgb 150 150 150 in
  let jeu = init_jeu 5 (larg*haut /2) larg haut tc
    squel (start jeu) (stop jeu) (key la jeu) mouse ex
val go : int -> int -> int -> unit = <fun>
```

Résumé

Ce chapitre a présenté les notions de base de la programmation graphique et de la programmation par événements en utilisant la bibliothèque Graphics de la distribution Objective CAML. Après avoir détaillé les éléments graphiques de base (couleur, tracé, remplissage, texte et bitmap) l'animation de ceux-ci a été abordée. Le mécanisme de gestion d'événements de Graphics a été ensuite détaillé ce qui a permis d'introduire une méthode générale de gestion d'interaction avec l'utilisateur mettant en jeu un modèle de programmation par événements. Pour améliorer l'interaction et proposer au programmeur des composants graphiques interactifs, il a été développé une nouvelle bibliothèque, appelée Upi, facilitant la construction d'interfaces graphiques. Cette bibliothèque a été utilisée pour écrire l'interface de la calculatrice impérative.

Pour en savoir plus

Bien qu'en programmation graphique et événementielle, le style de programmation soit impératif, il est non seulement possible, mais bien souvent utile, d'introduire des opérateurs plus fonctionnels de manipulation d'objets graphiques. Un bon exemple vient de l'utilisation de la bibliothèque MLgraph,

Lien: <http://www.pps.jussieu.fr/~cousinea/MLgraph/mlgraph.html>

qui plante le modèle graphique de PostScript, et propose des opérateurs fonctionnels de manipulation des images. Elle est décrite dans [CCS96] et utilisée abondamment dans [CM95] pour le placement optimisé des arbres et pour la construction de dessins dans le style d'Escher.

Une caractéristique intéressante de la bibliothèque Graphics est d'être portable pour les interfaces graphiques des systèmes Windows, MacOS et Unix. Cette notion de bitmap virtuel se retrouve dans plusieurs langages comme Le_Lisp ou plus récemment en Java. Malheureusement en Objective CAML la bibliothèque Graphics ne possède pas de composant interactif pour la construction d'interfaces. Une des applications décrites dans la partie II de ce livre contient les premières briques de la bibliothèque Upi. Celle-ci s'inspire de l'Abstract Windowing Toolkit des premières versions de Java. On s'aperçoit qu'il est relativement aisé d'étendre des fonctionnalités de cette bibliothèque grâce aux valeurs fonctionnelles du langage. Le chapitre 16 compare d'ailleurs l'adaptation de la programmation objet et de la programmation fonctionnelle et modulaire pour la construction d'interfaces graphiques. L'exemple d'Upi est fonctionnel et impératif, mais il est aussi possible de n'utiliser que le style fonctionnel. C'est typiquement le cas pour les langages fonctionnels purs. On peut citer les systèmes Fran et Fudget, développés en Haskell et dérivés. Le système Fran permet de construire des animations 2D, 3D interactives, c'est-à-dire avec événements entre objets animés ou utilisateurs.

Lien: <http://www.research.microsoft.com/~conal/fran/>

La bibliothèque Fudget est une bibliothèque pour la construction d'interfaces graphiques.

Lien: <http://www.cs.chalmers.se/ComputingScience/Research/Functional/Fudgets/>

Une des difficultés, quand on veut programmer une interface graphique pour son application, est de savoir laquelle choisir parmi les nombreux outils existants. Il ne suffit pas de déterminer le langage et le système pour fixer le choix de l'outil. Pour Objective CAML, il en existe plusieurs plus ou moins finalisés :

- l'encapsulation de la libX, pour X-Windows;
- la librt, une bibliothèque pour X-Windows;
- ocamltk, adaptation de Tcl/Tk, portable;
- mlgtk, adaptation de Gtk, portable.

On trouve les liens vers ces développements à la « bosse du chameau » :

Lien: <http://caml.inria.fr/hump.html>

Enfin nous avons parlé uniquement de la programmation en 2D. La tendance est à l'augmentation d'une dimension. Que cela soit le modèle VRML, ou l'extension Java-3D, les langages fonctionnels doivent aussi répondre à cette nécessité. En fonctionnel pur, le système Fran offre des possibilités intéressantes d'interaction entre sprites. Plus proche d'Objective CAML on peut utiliser la bibliothèque VRcAML ou l'environnement de développement SCOL.

À la manière de MLgraph a été développée la bibliothèque VRcAML qui intègre une partie du modèle graphique de VRML à Objective CAML.

Lien: <http://www.pps.jussieu.fr/~emmanuel/Public/enseignement/VRcAML>

On peut donc construire une scène 3D animée. Le résultat produit alors un fichier VRML directement visualisable.

Toujours dans la lignée de Caml, le langage SCOL, est un langage fonctionnel de communication muni d'importantes bibliothèques de manipulation 2D et 3D agrémenté d'un environnement de développement pour le non-informaticien.

Lien: <http://www.cryo-networks.com>

L'intérêt du langage SCOL et de son environnement de développement est de pouvoir réaliser des applications réparties, en client-serveur, facilitant la fabrication de sites Internet. On présente la programmation distribuée en Objective CAML au chapitre 20.

Applications

Introduction

L'intérêt d'un langage informatique par rapport à un autre provient de la facilité de développer des applications dont l'exécution est sûre et dont la maintenance est aisée. La première partie de cet ouvrage consacrée à la présentation du langage Objective CAML va logiquement se clore avec la réalisation de plusieurs applications.

La première application propose l'implantation de quelques fonctions permettant l'expression de requêtes sur une base de données. L'accent est mis sur l'utilisation d'un style de programmation fonctionnel et sur les manipulations de listes. On peut ainsi fournir à l'utilisateur un ensemble de fonctions permettant d'exprimer et d'évaluer facilement ses requêtes en utilisant directement le langage Objective CAML. On montre au programmeur comment il peut facilement fournir à l'utilisateur la plupart des outils d'interrogation dont ce dernier peut avoir besoin.

La deuxième application est un interprète pour un petit BASIC¹. Il est à noter que ce type de langage impératif a fait le succès des premiers micro-ordinateurs. Vingt ans après, cela semble fort simple à réaliser. Bien que BASIC soit un langage impératif, l'implantation de cet interprète utilisera plutôt la partie fonctionnelle d'Objective CAML, en particulier pour l'évaluation des instructions. Néanmoins les parties d'analyses lexicale et syntaxique du langage utilisent une structure physiquement modifiable.

La troisième application est un jeu à un joueur, le Démineur, bien connu car il est incorporé à l'installation standard des systèmes Windows. Le but du jeu est de découvrir un ensemble de mines

cachées en envoyant des sondes qui indiquent le nombre de mines autour d'une case déterminée. L'implantation utilise plutôt la partie impérative du langage, car la structure de données sous-jacente est un tableau à deux dimensions qui sera physiquement modifié à chaque coup joué. Cette application utilise le module Graphics pour le dessin du jeu et pour gérer l'interaction avec le joueur. Néanmoins la détection des cases à découvrir sera réalisée dans un style plus fonctionnel.

Cette dernière application fera appel à des fonctions du module Graphics décrit au chapitre 5 (voir page ??) ainsi qu'à quelques fonctions des modules Random et Sys (voir chapitre 8, pages ?? et ??).

Requêtes sur une base de données

La réalisation d'une base de données, de son interface et de son langage de requêtes est un projet trop ambitieux pour le cadre de ce livre et pour les connaissances en Objective CAML du lecteur rendu à ce point. Pourtant, en restreignant le problème et en profitant au mieux des possibilités offertes par la programmation fonctionnelle, nous allons pouvoir réaliser un outil intéressant pour le traitement des requêtes. Nous verrons, en particulier, comment utiliser les itérateurs ainsi que l'application partielle pour formuler et exécuter des requêtes. Nous montrerons également l'utilisation d'un type de données encapsulant des valeurs fonctionnelles.

Tout au long de cette application, nous utiliserons en guise d'exemple une base de données rassemblant les informations sur les adhérents d'une association. Elle est supposée être contenue dans le fichier association.dat.

Format des données

La plupart des logiciels classiques de base de données utilisent un format dit « propriétaire » pour stocker les informations qu'ils manipulent. Cependant, il est généralement possible de les sauver sous un format texte auquel on peut donner la forme suivante :

- la base est une suite de *fiches* séparées par des retour-chariot ;
- chaque fiche est une suite de *champs* séparés par un caractère quelconque que nous supposons ici être le caractère ' : ' ;
- un champ est une chaîne de caractères qui ne contient ni retour-chariot, ni le caractère ' : ' ;
- la première fiche est la liste des noms associés aux champs séparés par ' | '.

Le fichier de l'association débute par :

```
Num|Nom|Prenom|Adresse|Tel|Email|Pref|Date|Montant
0:Chailloux:Emmanuel:Université P6:0144274427:ec@lip6.fr
1:Manoury:Pascal:Laboratoire PPS::pm@lip6.fr:adr:03.03.1
2:Pagano:Bruno:Cristal:0139633963::adr:25.12.1998:150.00
3:Baro:Sylvain::0144274427:baro@pps.fr:mail:01.03.1999:5
```

La signification des champs est la suivante :

- Num est le numéro d'adhérent ;
- Nom, Prenom, Adresse, Tel et Email parlent d'eux-mêmes ;
- Pref indique comment l'adhérent désire recevoir les informations : par courrier postal (adr), par courrier électronique (mail) ou par téléphone (tel) ;
- Date et Montant sont respectivement la date et le montant de la dernière cotisation perçue.

Il nous faut dès à présent choisir la représentation que le programme adoptera pour stocker une base de données. Une alternative s'offre : soit une liste de fiches, soit un tableau de fiches. La liste présente l'avantage d'être aisément modifiable : l'ajout et la suppression d'une fiche sont des opérations simples. Le tableau, pour sa part, offre un accès en temps constant à n'importe quelle fiche. Comme notre but est de travailler sur l'ensemble des fiches et non pas sur certaines en particulier, chaque requête devra traiter la totalité des fiches. On peut donc choisir une structure de liste. La même alternative se pose pour les fiches : listes ou tableaux de chaînes de caractères ? Cette fois-ci, la réponse est inverse car, d'une part le format d'une fiche est figé pour toute la base, il n'est donc pas question de rajouter de nouveaux champs ; d'autre part, selon les traitements envisagés, seuls certains champs seront utiles, il est donc important d'y accéder rapidement.

La solution la plus naturelle pour une fiche serait d'avoir recours à un tableau indexé par le nom des champs. Un tel type n'étant pas possible en Objective CAML, nous allons le remplacer par un tableau classique (indexé par des entiers) et une fonction associant à un champ, l'indice du tableau lui correspondant.

```
# type data_card = string array ;;  
# type data_base = { card_index : string -> int ; data :
```

L'accès au champ de nom *n* d'une fiche *dc* de la base *db* est réalisé par la fonction :

```
# let field db n (dc:data_card) = dc.(db.card_index n) ;  
val field : data_base -> string -> data_card -> string =
```

Le type de *dc* a été forcé à *data_card* pour contraindre la fonction *field* à n'accepter que des tableaux de chaînes de caractères et non des tableaux quelconques.

Voici un petit exemple en guise d'illustration :

```
# let base_ex =  
  { data = [ [|"Chailloux"; "Emmanuel"|] ; [|"Manoury"; "Pascal"|] ] ;  
    card_index = function "Nom"->0 | "Prenom"->1 | _->raise Not_found }  
val base_ex : data_base =  
  { card_index=<fun>;  
    data=[ [|"Chailloux"; "Emmanuel"|] ; [|"Manoury"; "Pascal"|] ] ;  
# List.map (field base_ex "Nom") base_ex.data ;;  
- : string list = ["Chailloux"; "Manoury"]
```

L'expression `field base_ex "Nom"` s'évalue comme une fonction qui prend une fiche et renvoie son champ "Nom". Par l'utilisation de `List.map`, cette fonction est exécutée sur chacune des fiches de la base *base_ex* et on obtient la liste des résultats : soit la liste des champs "Nom" de la base.

Cet exemple nous permet de voir comment nous souhaitons utiliser la fonctionnalité dans notre programme. Ici, l'application partielle de *field* nous permet de définir une fonction d'accès à un champ précis, utilisable sur un nombre quelconque de fiches. Cela nous

permet aussi de voir que notre implantation de cette même fonction `field` souffre d'un léger défaut car, alors que nous accédons toujours au même champ, son indice est tout de même recalculé à chaque fois. Pour cette raison, nous lui préférons l'implantation suivante :

```
# let field base name =  
  let i = base.card_index name in fun (card:data_card) ->  
  val field : data_base -> string -> data_card -> string =
```

Ici, après application de deux arguments, l'indice du champ est effectivement calculé et sert pour toutes les applications ultérieures.

Lecture d'une base depuis un fichier

Depuis Objective CAML, un fichier contenant une base est une suite de lignes. Notre travail va consister à lire chacune d'entre elles comme une chaîne de caractères puis à la découper en repérant les séparateurs et à construire d'une part les données et d'autre part la fonction d'indexation des champs.

Utilitaires de traitement de ligne

Nous nous dotons d'une fonction `split` qui découpe une chaîne de caractères selon un caractère séparateur. Cette fonction utilise la fonction `suffix` retournant le suffixe d'une chaîne `s` à partir d'une position `i`. On utilise pour cela trois fonctions prédéfinies :

- `String.length` donne la longueur d'une chaîne;
- `String.sub` retourne la sous-chaîne de `s` à partir de la position `i` de longueur `l`;
- `String.index_from` calcule dans la chaîne `s`, à partir de la position `n`, la position de la première occurrence du caractère `c`.

```
# let suffix s i = try String.sub s i ((String.length s)  
  with Invalid_argument("String.sub") -> "" ;;  
val suffix : string -> int -> string = <fun>  
# let split c s =
```

```

let rec split_from n =
try let p = String.index_from s n c
in (String.sub s n (p-n)) :: (split_from (p+1))
with Not_found -> [ suffix s n ]
in if s="" then [] else split_from 0 ;;
val split : char -> string -> string list = <fun>

```

La seule chose à noter dans l'implantation de ces fonctions est la gestion des exceptions, en particulier, l'exception `Not_found`.

Calcul de la structure *data_base*

Obtenir un tableau de chaînes à partir d'une liste ne pose pas de problème : le module `Array` fournit la fonction nécessaire (`of_list`). Le calcul de la fonction d'indice à partir d'une liste de noms de champs pourrait paraître plus compliqué mais le module `List` fournit tous les outils dont nous avons besoin.

Nous partons d'une liste de chaînes et nous devons obtenir une fonction qui à une chaîne associe un indice qui correspond à sa place dans la liste.

```

# let mk_index list_names =
  let rec make_enum a b = if a > b then [] else a::(make_
  let list_index = (make_enum 0 ((List.length list_names
  let assoc_index_name = List.combine list_names list_inde
  function name -> List.assoc name assoc_index_name ;;
val mk_index : 'a list -> 'a -> int = <fun>

```

Pour obtenir la fonction d'association entre noms de champ et indices, nous construisons la liste des indices que nous combinons avec la liste de noms. Nous obtenons une liste d'association de type *string * int list*. Pour chercher l'indice associé à un nom nous avons recours à la fonction `assoc` de la bibliothèque `List` qui est dédiée à cette tâche. Le résultat de `make_index` est une fonction qui prend un nom et appelle `assoc` avec ce nom sur la liste d'association précédemment construite.

Nous pouvons maintenant définir la fonction de lecture des fichiers

contenant une base au format imposé.

```
# let read_base name_file =  
  let channel = open_in name_file in  
  let split_line = split ':' in  
  let list_names = split '|' (input_line channel) in  
  let rec read_file () =  
    try  
      let data = Array.of_list (split_line (input_line channel))  
      data :: (read_file ())  
    with End_of_file -> close_in channel ; []  
  in  
    { card_index = mk_index list_names ; data = read_file () }  
val read_base : string -> data_base = <fun>
```

La lecture des enregistrements du fichier est réalisée par la fonction auxiliaire `read_file` qui opère récursivement sur le canal d'entrée. Le cas de base de la récurrence correspond à la fin du fichier signalée par l'exception `End_of_file`. Notons que nous profitons de ce cas retournant la liste vide pour refermer le canal.

Nous pouvons maintenant charger le fichier de notre association :

```
# let base_ex = read_base "association.dat" ;;  
val base_ex : data_base =  
  {card_index=<fun>;  
   data=  
    [|"0"; "Chailloux"; "Emmanuel"; "Universit\233 P6"; "0  
    "ec@lip6.fr"; "mail"; "25.12.1998"; "100.00"|];  
    [|"1"; "Manoury"; "Pascal"; "Laboratoire PPS"; ...|]; .
```

Principes généraux des traitements

La richesse et la complexité du traitement de l'ensemble des données d'une base sont proportionnelles à la richesse et à la complexité du langage de requêtes utilisé. Puisqu'ici, notre volonté est d'utiliser Objective CAML comme langage de requêtes, il n'y a a

priori pas de limite à l'expression des requêtes ! Cependant, nous souhaitons également fournir quelques outils simples de manipulation des fiches et de leurs données. Pour obtenir cette simplicité, il nous faut nécessairement limiter la trop grande richesse du langage Objective CAML en posant quelques objectifs et principes généraux de traitement.

L'objectif d'un traitement de données est d'obtenir ce que l'on appelle un état de la base. On peut décomposer la construction d'un état en trois étapes :

1. sélection, selon un critère donné, d'un ensemble des fiches ;
2. traitement individuel de chacune des fiches sélectionnées ;
3. traitement de l'ensemble des données collectées sur chacune des fiches.

Cette décomposition est illustrée à la figure 6.1.

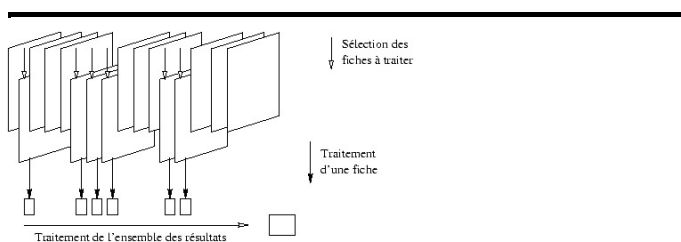


Figure 6.1 : décomposition d'une requête

Suivant cette décomposition, nous avons besoin de trois fonctions possédant les types suivants :

1. $(data_card \rightarrow bool) \rightarrow data_card\ list \rightarrow data_card\ list$
2. $(data_card \rightarrow 'a) \rightarrow data_card\ list \rightarrow 'a\ list$
3. $('a \rightarrow 'b \rightarrow 'b) \rightarrow 'a\ list \rightarrow 'b \rightarrow 'b$

Objective CAML nous fournit trois fonctions d'ordre supérieur, ou itérateurs, présentés à la page ??, qui répondent à notre spécification :

```
# List.find_all ;;
- : ('a -> bool) -> 'a list -> 'a list = <fun>
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# List.fold_right ;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Nous pourrions les utiliser pour réaliser les trois étapes de construction d'un état lorsque nous aurons fixé la valeur de leur argument fonctionnel.

Pour certaines requêtes particulières, on utilisera également :

```
# List.iter ;;
- : ('a -> unit) -> 'a list -> unit = <fun>
```

En effet, si le traitement d'un ensemble d'informations se limite à leur affichage, il n'y a pas réellement de valeur à calculer.

Nous allons, dans les paragraphes suivants, voir comment définir des fonctions exprimant les critères de sélection ainsi que quelques traitements simples. Nous conclurons ce chapitre par un petit exemple mettant en oeuvre ces fonctions selon le principe énoncé ci-dessus.

Critères de sélection

Dans la pratique, la fonction booléenne qui exprime le critère de sélection d'une fiche est obtenue par combinaison booléenne de propriétés données sur tout ou partie des champs de la fiche. Chaque champ d'une fiche, quoique donné comme une chaîne de caractères, peut être porteur d'une information d'un autre type : un flottant, une date, etc.

Critères de sélection sur un champ

La composante d'un critère de sélection concernant un champ

particulier sera, en général, obtenue à partir d'une fonction de type *data_base -> 'a -> string -> data_card -> bool*. Le paramètre de type *'a* correspond au type de l'information contenue dans le champ. L'argument de type *string* correspond au nom du champ.

Informations de type chaînes de caractères

Nous définissons deux tests simples sur ces types de champ : l'égalité avec une autre chaîne et la non vacuité.

```
# let eq_sfield db s n dc = (s = (field db n dc)) ;;
val eq_sfield : data_base -> string -> string -> data_card -> bool
# let nonempty_sfield db n dc = (" " <> (field db n dc))
val nonempty_sfield : data_base -> string -> data_card -> bool
```

Informations de type flottant

Pour réaliser des tests sur des informations de type flottant, il suffit simplement d'opérer la conversion d'une valeur de type *string* représentant un nombre décimal en une valeur de type *float*. Voici quelques exemples obtenus à partir d'une fonction générique *tst_ffield* :

```
# let tst_ffield r db v n dc = r v (float_of_string (field db n dc))
val tst_ffield : ('a -> float -> 'b) -> data_base -> 'a -> string -> data_card -> bool
# let eq_ffield = tst_ffield ( = ) ;;
# let lt_ffield = tst_ffield ( < ) ;;
# let le_ffield = tst_ffield ( <= ) ;;
(* etc. *)
```

Ces trois fonctions sont de type *:data_base -> float -> string -> data_card -> bool*.

Les dates

Ce type d'information est un peu plus complexe à traiter. Il dépend de la représentation de la date dans la base et réclame de définir un codage des comparaisons de dates.

Nous fixons qu'une date est représentée dans une fiche par une chaîne de caractères au format `jj.mm.aaaa`. Pour les besoins des comparaisons que nous souhaitons obtenir,

nous enrichissons ce format en autorisant le remplacement du jour, du mois ou de l'année par le caractère souligné ('_'). Les dates sont comparées selon l'ordre lexicographique sur des triplets d'entiers de la forme (*année, mois, jour*). Pour pouvoir exprimer des requêtes telles : `< < est antérieure à juillet 1998 >`, on utilisera le *motif de date* : `"_.07.1998"`. La comparaison d'une date avec un motif sera réalisée par la fonction `tst_dfield` qui analysera le motif pour en extraire la fonction de comparaison *ad hoc*. La définition de cette fonction générique de test sur les dates nécessite un certain nombre de fonctions auxiliaires.

On se donne deux fonctions de conversion de dates (`ints_of_string`) et de motifs de dates (`ints_of_dpat`) en triplets d'entiers. Le caractère '_' d'un motif sera remplacé par l'entier 0 :

```
# let split_date = split '.' ;;
val split_date : string -> string list = <fun>
# let ints_of_string d =
  try match split_date d with
    [j;m;a] -> [int_of_string a; int_of_string m; int_of_string j]
    | _ -> failwith "Bad date format"
  with Failure("int_of_string") -> failwith "Bad date format"
val ints_of_string : string -> int list = <fun>

# let ints_of_dpat d =
  let int_of_stringpat = function "_" -> 0 | s -> int_of_string s
  in try match split_date d with
    [j;m;a] -> [ int_of_stringpat a; int_of_stringpat m;
      int_of_stringpat j ]
    | _ -> failwith "Bad date format"
  with Failure("int_of_string") -> failwith "Bad date pattern"
val ints_of_dpat : string -> int list = <fun>
```

Étant donnée une relation *r* sur les entiers, on écrit la fonction d'application du test. C'est un codage de l'ordre lexicographique prenant en compte la valeur particulière 0 que l'on ignore :

```
# let rec app_dtst r d1 d2 = match d1, d2 with
  [] , [] -> false
  | (0::d1) , (_::d2) -> app_dtst r d1 d2
  | (n1::d1) , (n2::d2) -> (r n1 n2) || ((n1 = n2) && (ap
  | _, _ -> failwith "Bad date pattern or format" ;;
val app_dtst : (int -> int -> bool) -> int list -> int list -> bool
```

On définit enfin la fonction générique *tst_dfield* qui prend comme arguments une relation *r*, une base de données *db*, un motif *dp*, un nom de champ *nm* et une fiche *dc*. Cette fonction vérifie que le motif et le champ extrait de la fiche satisfont la relation.

```
# let tst_dfield r db dp nm dc =
  r (ints_of_dpat dp) (ints_of_string (field db nm dc)) ;
val tst_dfield :
  (int list -> int list -> 'a) ->
  data_base -> string -> string -> data_card -> 'a = <fun
```

On l'applique alors à trois relations.

```
# let eq_dfield = tst_dfield (=) ;;
# let le_dfield = tst_dfield (<=) ;;
# let ge_dfield = tst_dfield (>=) ;;
```

Ces trois fonctions sont de type : *data_base -> string -> string -> data_card -> bool*.

Composition de critères

Les tests que nous avons définis ci-dessus ont pour trois premiers arguments une base de données, une valeur et le nom d'un champ. Lorsque nous formulons des requêtes particulières, les valeurs de ces trois arguments sont connues. Lorsque l'on travaille sur la base *base_ex*, le test << est antérieure à juillet 1998 >> s'écrit

```
# ge_dfield base_ex "_07.1998" "Date" ;;
- : data_card -> bool = <fun>
```

En pratique, un test est donc une fonction de type *data_card -> bool*. Nous voulons obtenir des combinaisons booléennes des résultats de telles fonctions appliquées à une même fiche. Nous nous donnons à cette fin l'itérateur ;

```
# let fold_funs b c fs dc =
  List.fold_right (fun f -> fun r -> c (f dc) r) fs b ;;
val fold_funs : 'a -> ('b -> 'a -> 'a) -> ('c -> 'b) list -> 'a
```

Où b est la valeur de base, la fonction c sera le connecteur booléen, fs la liste des fonctions de test sur un champ et dc une fiche.

On obtient la conjonction et la disjonction d'une liste de tests par :

```
# let and_fold fs = fold_funs true (&) fs ;;
val and_fold : ('a -> bool) list -> 'a -> bool = <fun>
# let or_fold fs = fold_funs false (or) fs ;;
val or_fold : ('a -> bool) list -> 'a -> bool = <fun>
```

Par commodité, on définit également la négation d'un test :

```
# let not_fun f dc = not (f dc) ;;
val not_fun : ('a -> bool) -> 'a -> bool = <fun>
```

On peut, par exemple, utiliser ces combinateurs pour définir une fonction de sélection d'une fiche dont le champ date est compris dans un intervalle donné :

```
# let date_interval db d1 d2 =
  and_fold [(le_dfield db d1 "Date"); (ge_dfield db d2 "Date")] not_fun ;;
val date_interval : data_base -> string -> string -> bool = <fun>
```

Traitements et calculs

Il est difficile de prévoir ce que peut être a priori le traitement d'une fiche ou de l'ensemble des données issues de ce traitement. Néanmoins, on peut considérer deux grandes classes de tels traitements : un calcul numérique ou le formatage de données pour impression. Prenons un exemple de chacun des cas envisagés.

Formatage

On désire préparer pour l'impression une ligne contenant l'identité d'un adhérent suivie d'un certain nombre d'informations.

On commence par se donner l'opération inverse du découpage d'une chaîne selon un caractère donné :

```
# let format_list c =  
  let s = String.make 1 c in  
  List.fold_left (fun x y -> if x="" then y else x^s^y) "" c  
val format_list : char -> string list -> string = <fun>
```

Afin de construire la liste des champs d'information, on se donne la fonction `extract` qui extrait d'une fiche un ensemble de champs définis par une liste de noms :

```
# let extract db ns dc =  
  List.map (fun n -> field db n dc) ns ;;  
val extract : data_base -> string list -> data_card -> string list
```

On écrit enfin la fonction de formatage d'une ligne :

```
# let format_line db ns dc =  
  (String.uppercase (field db "Nom" dc))  
  ^ " " ^ (field db "Prenom" dc)  
  ^ "\t" ^ (format_list '\t' (extract db ns dc))  
  ^ "\n" ;;  
val format_line : data_base -> string list -> data_card -> string
```

L'argument `ns` est la liste des champs d'information désirés. Les champs d'information sont séparés par le caractère de tabulation ('`\t`') et la ligne se termine par un retour-chariot.

On obtient l'affichage de la liste des noms et prénoms des adhérents par :

```
# List.iter print_string (List.map (format_line base_ex
CHAILLOUX Emmanuel
MANOURY Pascal
PAGANO Bruno
BARO Sylvain
- : unit = ())
```

Calcul numérique

On veut calculer la somme des cotisations perçues pour un ensemble de fiches. On obtient facilement ce chiffre en composant l'extraction et la conversion du champ correspondant avec une fonction d'addition. Pour alléger l'écriture, on définit un opérateur infixe de composition :

```
# let (++) f g x = g (f x) ;;
val ++ : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>
```

On utilise cet opérateur de composition dans la définition suivante :

```
# let total db dcs =
  List.fold_right ((field db "Montant") ++ float_of_string) dcs 0.0
val total : data_base -> data_card list -> float = <fun>
```

On peut l'appliquer à la totalité de la base :

```
# total base_ex base_ex.data ;;
- : float = 450
```

Un exemple

Nous donnons, pour conclure ce chapitre un petit exemple

d'application des principes exposés dans les paragraphes précédents.

Nous envisageons deux types de requêtes sur notre base :

- l'édition de deux listes contenant chacune l'identité de l'adhérent, puis, selon ses préférences, son adresse postale ou son adresse électronique.
- l'édition d'un état des cotisations entre deux dates données. Cet état contiendra la liste des nom, prénom, date et montant des cotisations ainsi que le total de ces dernières.

Listes d'adresses

Pour obtenir ces listes, on sélectionne d'abord les fiches pertinentes selon la valeur du champ "Pref" puis on utilise la fonction de formatage `format_line` :

```
# let adresses_postales db =  
  let dcs = List.find_all (eq_sfield db "adr" "Pref") db.  
  List.map (format_line db ["Adresse"]) dcs ;;  
val adresses_postales : data_base -> string list = <fun>  
  
# let adresses_electroniques db =  
  let dcs = List.find_all (eq_sfield db "mail" "Pref") db.  
  List.map (format_line db ["Email"]) dcs ;;  
val adresses_electroniques : data_base -> string list =
```

État des cotisations

Le calcul de l'état des cotisations procède toujours selon le même esprit : sélection puis traitement. Mais le traitement est dédoublé en formatage des lignes puis calcul du total.

```
# let etat_cotisations db d1 d2 =  
  let dcs = List.find_all (date_interval db d1 d2) db.dat  
  let ls = List.map (format_line db ["Date"; "Montant"]) d  
  let t = total db dcs in  
  ls, t ;;  
val etat_cotisations : data_base -> string -> string ->
```

<fun>

Le résultat de cette requête est un couple contenant la liste des chaînes d'information et le montant cumulé des cotisations.

Programme principal

Le programme principal de l'application est essentiellement une boucle d'interaction avec l'utilisateur qui affiche le résultat de la requête demandée par menu. On y retrouve naturellement un style de programmation impératif, même si l'affichage des résultats fait appel à un itérateur.

```
# let main() =  
  let db = read_base "association.dat" in  
  let fin = ref false in  
  while not !fin do  
    print_string " 1: liste des adresses postales\n";  
    print_string " 2: liste des adresses électroniques\n";  
    print_string " 3: cotisations\n";  
    print_string " 0 sortie\n";  
    print_string "Votre choix : ";  
    match read_int() with  
    0 -> fin := true  
    | 1 -> (List.iter print_string (adresses_postales db))  
    | 2 -> (List.iter print_string (adresses_electroniques  
    | 3  
    -> (let d1 = print_string "Date de début : "; read_line()  
    let d2 = print_string "Date de fin : "; read_line() in  
    let ls, t = etat_cotisations db d1 d2 in  
    List.iter print_string ls;  
    print_string "Total : "; print_float t; print_newline())  
    | _ -> ())  
  done;  
  print_string "bye\n" ;;  
val main : unit -> unit = <fun>
```

Cet exemple sera repris au chapitre [21](#) pour le munir d'une interface utilisant un navigateur WEB.

Pour en faire plus

Une extension naturelle de notre exemple serait de munir la base d'une indication de type associée à chaque champ de la base. Cette information pourrait alors être exploitée pour définir des opérateurs de comparaison généraux de type *data_base* -> 'a' -> *string* -> *data_card* -> *bool* où le nom du champ (troisième argument) permettrait d'aiguiller sur les bonnes fonctions de conversion et de test.

Interprète BASIC

L'application présentée dans cette section est un interprète de programmes Basic. C'est donc un programme sachant exécuter d'autres programmes écrits en Basic. Bien entendu, nous ne décrirons qu'un langage restreint, il ne contient que les instructions suivantes :

- **PRINT** *expression*

Affiche le résultat de l'évaluation de l'expression.

- **INPUT** *variable*

Affiche un *prompt* à l'écran (?), attend qu'un entier soit entré au clavier, et affecte la variable avec cette valeur.

- **LET** *variable* = *expression*

Affecte la variable avec le résultat de l'évaluation de l'expression.

- **GOTO** *numéro de ligne*

Reprend l'exécution à la ligne

mentionnée.

- **IF condition THEN** *numéro de ligne*

Reprend l'exécution à la ligne mentionnée si la *condition* est vraie.

- **REM** *chaîne de caractères quelconque*

Commentaire sur une ligne.

Chaque ligne d'un programme Basic est étiquetée par son numéro de ligne et ne contient qu'une seule instruction. Par exemple, un programme calculant puis affichant la factorielle d'un entier entré au clavier s'écrit :

```
5  REM  entree de l'argument
10 PRINT " factorielle de : "
20 INPUT A
30 LET B = 1
35 REM debut de la boucle
40 IF A <= 1 THEN 80
50 LET B = B * A
60 LET A = A - 1
70 GOTO 40
75 REM le resultat est affiche
80 PRINT B
```

Nous souhaitons de surcroît réaliser un mini-éditeur fonctionnant sur le principe d'une boucle d'interaction. Il devra rendre possible l'ajout de nouvelles lignes, l'affichage d'un programme et l'évaluation. L'exécution du programme précédent est lancée par la commande RUN. Voici un exemple d'évaluation de ce programme :

```
> RUN
  factorielle de : ? 5
120
```

Nous décomposons la réalisation de cet évaluateur en plusieurs parties relativement distinctes :

Description de la syntaxe abstraite

: il faut définir des types de données permettant de représenter les programmes Basic ainsi que leurs constituants (lignes, instructions, expressions, etc.).

Affichage du programme

: cette partie consiste à transformer la représentation interne des programmes Basic en chaînes de caractères afin de les visualiser.

Analyse lexicale et analyse syntaxique

: ces deux parties réalisent l'opération inverse, à savoir transformer une chaîne de caractères en la représentation interne d'un programme Basic (sa syntaxe abstraite).

Évaluation

: le coeur de l'évaluateur gouvernant l'exécution du programme. Nous verrons qu'un langage fonctionnel comme Objective CAML est particulièrement bien adapté à ce genre de problème.

Boucle d'interaction

: elle met en oeuvre ce qui précède.

Syntaxe abstraite

La figure 6.2 présente la syntaxe concrète du Basic que nous allons implanter par une grammaire sous la forme BNF. Cette façon de présenter la syntaxe d'un langage est décrite au chapitre 11, page ??.

Op_Unaire	::=	- !
Op_Binaire	::=	+ - * / %
		= < >
		<= >=
		< >
		& ' '

Expression	::=	<i>entier</i>
		<i>variable</i>
		"chaîne de caractères"
		Op_Unaire
		Expression
		Expression
		Op_Binaire
		Expression
		(Expression)
Instruction	::=	REM <i>chaîne de caractères</i>
		GOTO <i>entier</i>
		LET <i>variable</i> =
		Expression
		PRINT
		Expression
		INPUT <i>variable</i>
		IF Expression
		THEN <i>entier</i>
Ligne	::=	<i>entier</i> Instruction
Programme	::=	Ligne
		Ligne
		Programme
Commande	::=	Ligne RUN
		LIST END

Figure 6.2 : Grammaire BASIC

Remarquons que la définition des expressions ne garantit pas qu'une expression bien formée soit évaluable. Par exemple, $1 + \text{"hello"}$ est une expression et pourtant il n'est pas possible de l'évaluer. Ce choix délibéré permet de simplifier la syntaxe abstraite du langage Basic et de raccourcir l'analyse syntaxique. Le prix à payer est la possibilité que des programmes Basic syntaxiquement corrects puissent produire une erreur d'exécution due à une incohérence de type.

La définition des types de données Objective CAML pour cette syntaxe abstraite découle immédiatement de la définition de la syntaxe concrète en utilisant un type somme :

```
# type op_unr = OPPOSE | NON ;;
# type op_bin = PLUS | MOINS | MULT | DIV | MOD
| EGAL | INF | INFEQ | SUP | SUPEQ | DIFF
| ET | OU ;;
# type expression =
  ExpInt of int
| ExpVar of string
| ExpStr of string
| ExpUnr of op_unr * expression
| ExpBin of expression * op_bin * expression ;;
# type instruction =
  Rem of string
| Goto of int
| Print of expression
| Input of string
| If of expression * int
| Let of string * expression ;;
# type ligne = { num : int ; inst : instruction } ;;
# type program = ligne list ;;
```

On définit les éléments de syntaxe abstraite pour les commandes du mini-éditeur de programme :

```
# type phrase = Ligne of ligne | List | Run | End ;;
```

Il est d'usage d'alléger l'écriture des expressions arithmétiques en autorisant le programmeur à ne pas mentionner toutes les parenthèses. Par exemple, l'expression $1 + 3 * 4$ est habituellement interprétée comme $1 + (3 * 4)$. On attribue, à cette fin, un entier à chacun des opérateurs du langage :

```
# let priority_ou = function NON -> 1 | OPPOSE -> 7
let priority_ob = function
  MULT | DIV -> 6
| PLUS | MOINS -> 5
| MOD -> 4
| EGAL | INF | INFEQ | SUP | SUPEQ | DIFF -> 3
```



```
| ET | OU -> 2 ;;
val priority_ou : op_unr -> int = <fun>
val priority_ob : op_bin -> int = <fun>
```

Ces entiers indiquent ce que l'on appelle la priorité des opérateurs. Nous verrons comment elle est utilisée pour l'affichage des programmes et l'analyse syntaxique.

Affichage des programmes

Pour afficher un programme contenu en mémoire, il faut savoir transformer une ligne de programme, sous forme de syntaxe abstraite, en une chaîne de caractères.

La conversion des opérateurs est immédiate :

```
# let pp_opbin = function
  PLUS -> "+" | MULT -> "*" | MOD -> "%" | MOINS -> "-"
  | DIV -> "/" | EGAL -> "=" | INF -> "<"
  | INFEQ -> "<=" | SUP -> ">"
  | SUPEQ -> ">=" | DIFF -> "<>" | ET -> "&" | O
  let pp_opunr = function OPPOSE -> "-" | NON -> "!" ;;
val pp_opbin : op_bin -> string = <fun>
val pp_opunr : op_unr -> string = <fun>
```

L'affichage des expressions tiendra compte des règles de priorité pour engendrer un parenthésage allégé. Par exemple, on ne parenthèse une sous-expression à droite d'un opérateur que lorsque son opérateur principal a une priorité inférieure à l'opérateur principal de l'expression entière. De plus, les opérateurs arithmétiques sont associatifs à gauche, c'est-à-dire que l'expression 1-2-3 est implicitement parenthésée comme (1-2)-3.

Pour ce faire, nous avons recours à deux fonctions auxiliaires ppg et ppd pour traiter respectivement les sous-arbres gauches et les sous-arbres droits. Ces fonctions prennent deux paramètres, d'une part l'arbre d'expression à afficher mais aussi la priorité de l'opérateur qui est en amont de cet arbre afin de décider s'il y a lieu de

parenthéser l'expression. Nous distinguons les sous-arbres gauches des sous-arbres droits pour traiter l'associativité des opérateurs. En cas d'égalité de priorité des opérateurs on ne mettra pas de parenthèse pour l'opérande gauche alors qu'elles sont indispensables dans le cas de l'opérande droit comme dans les expressions $1-(2-3)$ et $1 + (2 + 3)$.

On considère l'arbre initial comme un sous-arbre gauche sous un opérateur de priorité minimale (0). Voici la fonction `pp_expression` d'affichage des expressions :

```
# let parenthese x = "(" ^ x ^ ")";;
val parenthese : string -> string = <fun>
# let pp_expression =
  let rec ppg pr = function
    ExpInt n -> (string_of_int n)
    | ExpVar v -> v
    | ExpStr s -> "\"" ^ s ^ "\""
    | ExpUnr (op,e) ->
      let res = (pp_opunr op)^(ppg (priority_of op) e)
      in if pr=0 then res else parenthese res
    | ExpBin (e1,op,e2) ->
      let pr2 = priority_of op
      in let res = (ppg pr2 e1)^(pp_opbin op)^(ppd pr2 e2)
      (* parenthèse si la priorité n'est pas supérieure *)
      in if pr2 >= pr then res else parenthese res
      and ppd pr exp = match exp with
      (* les sous-arbres droits ne diffèrent *)
      (* que pour les opérateurs binaires *)
      ExpBin (e1,op,e2) ->
        let pr2 = priority_of op
        in let res = (ppg pr2 e1)^(pp_opbin op)^(ppd pr2 e2)
        in if pr2 > pr then res else parenthese res
        | _ -> ppg pr exp
      in ppg 0 ;;
val pp_expression : expression -> string = <fun>
```

L'affichage des instructions utilise la fonction précédente sur les expressions. L'affichage d'une ligne ajoutera un numéro de ligne devant l'instruction.

```
# let pp_instruction = function
  Rem s      -> "REM " ^ s
  | Goto n    -> "GOTO " ^ (string_of_int n)
  | Print e   -> "PRINT " ^ (pp_expression e)
  | Input v   -> "INPUT " ^ v
  | If (e,n)  -> "IF "^(pp_expression e)^" THEN "^(string_of_int n)^"
  | Let (v,e) -> "LET " ^ v ^ " = " ^ (pp_expression e)
val pp_instruction : instruction -> string = <fun>
# let pp_ligne l = (string_of_int l.num) ^ " " ^ (pp_instruction l.instr)
val pp_ligne : ligne -> string = <fun>
```

Analyse lexicale

Les analyses lexicale et syntaxique effectuent la transformation inverse de l'affichage. Elles reçoivent une chaîne de caractères et construisent un arbre de syntaxe. L'analyse lexicale découpe le texte d'une ligne d'instruction en unités lexicales indépendantes appelées lexèmes, dont le type Objective CAML suit :

```
# type lexeme = Lint of int
  | Lident of string
  | Lsymbol of string
  | Lstring of string
  | Lfin ;;
```

Nous avons rajouté un lexème particulier pour marquer la fin d'une expression : `Lfin`. Il ne fait pas partie de la chaîne analysée, mais sera engendré par la fonction d'analyse lexicale (voir fonction `lexer`, page ??).

La chaîne que l'on souhaite analyser est conservée dans un enregistrement contenant un champ modifiable indiquant l'indice de la partie de la chaîne restant à parcourir. La taille de la chaîne étant une information qui sera nécessaire en de multiples occasions et qui ne varie pas, nous la stockons dans l'enregistrement :

```
# type chaine_lexer = {chaine:string; mutable courant:int}
```

Cette représentation permet de définir l'analyse lexicale d'une chaîne comme l'application d'une fonction à une valeur de type *chaîne_lexer* et rendant une valeur de type *lexeme*. La modification de l'indice de la chaîne restant à parcourir se fait par effet de bord.

```
# let init_lex s = { chaine=s; courant=0 ; taille=String.length s ; }
val init_lex : string -> chaîne_lexer = <fun>
# let avance cl = cl.courant <- cl.courant+1 ;;
val avance : chaîne_lexer -> unit = <fun>
# let avance_n cl n = cl.courant <- cl.courant+n ;;
val avance_n : chaîne_lexer -> int -> unit = <fun>
# let extrait pred cl =
  let st = cl.chaine and ct = cl.courant in
  let rec ext n = if n<cl.taille && (pred st.[n]) then ext (n+1) else ct in
  let res = ext ct
  in cl.courant <- res ; String.sub cl.chaine ct (res-ct)
val extrait : (char -> bool) -> chaîne_lexer -> string =
```

Les fonctions suivantes extraient un lexème de la chaîne et positionnent le caractère courant sur le prochain caractère à traiter. Les deux fonctions auxiliaires : *extrait_int* et *extrait_ident* extraient respectivement un entier et un identificateur.

```
# let extrait_int =
  let est_entier = function '0'..'9' -> true | _ -> false
  in function cl -> int_of_string (extrait est_entier cl)
let extrait_ident =
  let est_alpha_num = function
    'a'..'z' | 'A'..'Z' | '0' .. '9' | '_' -> true
    | _ -> false
  in extrait est_alpha_num ;;
val extrait_int : chaîne_lexer -> int = <fun>
val extrait_ident : chaîne_lexer -> string = <fun>
```

La fonction *lexer* utilise les deux fonctions précédentes pour extraire un lexème.

```
# exception LexerErreur ;;
exception LexerErreur
```

```

# let rec lexer cl =
  let lexer_char c = match c with
  | ' '
  | '\t'      -> avance cl ; lexer cl
  | 'a'..'z'
  | 'A'..'Z' -> Lident (extrait_ident cl)
  | '0'..'9' -> Lint (extrait_int cl)
  | '"'      -> avance cl ;
  let res = Lstring (extrait ((<>) '"') cl)
  in avance cl ; res
  | '+' | '-' | '*' | '/' | '%' | '&' | '|' | '!' | '=' |
  avance cl; Lsymbol (String.make 1 c)
  | '<'
  | '>'      -> avance cl;
  if cl.courant >= cl.taille then Lsymbol (String.make 1
  else let cs = cl.chaine.[cl.courant]
  in ( match (c,cs) with
    ('<','=') -> avance cl; Lsymbol "<="
    | ('>','=') -> avance cl; Lsymbol ">="
    | ('<','>') -> avance cl; Lsymbol "<>"
    | _         -> Lsymbol (String.make 1 c) )
  | _ -> raise LexerErreur
  in
  if cl.courant >= cl.taille then Lfin
  else lexer_char cl.chaine.[cl.courant] ;;
val lexer : chaine_lexer -> lexeme = <fun>

```

Le principe de la fonction `lexer` est des plus simples, elle filtre le caractère courant d'une chaîne et, suivant sa valeur, rend le lexème correspondant en ayant avancé le caractère courant de la chaîne jusqu'au caractère suivant. Cette approche est bien adaptée car mis à part pour deux caractères, des lexèmes différents peuvent se distinguer dès leur premier caractère. Dans le cas de '`<`' nous sommes par contre obligés d'examiner le caractère suivant pour savoir s'il s'agit de `=` ou de `>` ce qui produira un lexème différent. De même pour le caractère '`>`'.

Analyse syntaxique

L'unique difficulté de l'analyse syntaxique de notre langage provient des expressions. En effet, la connaissance du début de l'expression ne suffit pas pour déterminer sa structure. Supposons que nous ayons analysé la portion de chaîne $1 + 2 + 3$. Suivant ce qui suit, par exemple : $+ 4$ ou $* 4$, nous obtiendrons des arbres d'expression dont la structure concernant $1 + 2 + 3$ n'est pas la même (voir figure 6.3).

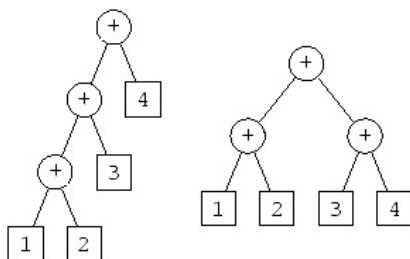


Figure 6.3 : Basic : exemple d'arbres de syntaxe abstraite

Cependant, la structure de l'arbre correspondant à $1 + 2$ est la même dans les deux cas. Nous pouvons donc la construire, comme le sort réservé à $+ 3$ est encore indéterminé, nous stockerons temporairement cette information jusqu'à ce que nous possédions suffisamment d'information pour pouvoir la traiter.

Nous allons procéder à la construction de l'arbre de syntaxe abstraite en ayant recours à un automate à pile proche de celui que construit l'outil *yacc* (voir page ??). Les lexèmes sont lus l'un après l'autre et sont empilés tant que l'information n'est pas suffisante pour construire l'expression. Lorsque nous savons décider de la structure des éléments d'expressions empilés, nous les retirons de la pile et les remplaçons par l'expression construite. Cette opération s'appelle réduction.

Le type des éléments empilés est le suivant :

```
# type exp_elem =
  Texp of expression      (* expression          *)
  | Tbin of op_bin        (* opérateur binaire *)
```

```
| Tunnr of op_unr          (* opérateur unaire *)
| Tpg          (* parenthèse gauche *) ;;
```

Remarquons que les parenthèses droites ne sont pas stockées car seules les parenthèses gauche sont significatives pour l'opération de réduction.

La figure 6.4 illustre le principe de l'utilisation de la pile pour l'analyse de l'expression $(1 + 2 * 3) + 4$. Le caractère surmontant la flèche est le caractère courant de la chaîne.

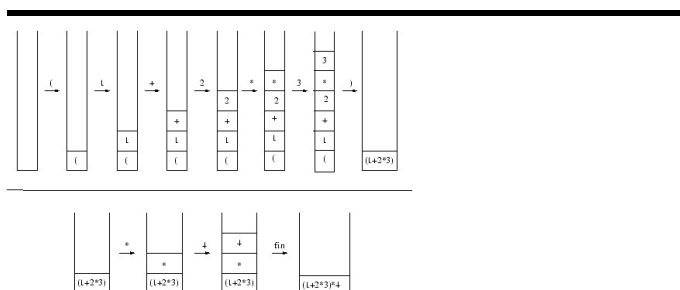


Figure 6.4 : Basic : exemple de construction d'un arbre de syntaxe abstraite

On définit l'exception suivante pour les erreurs de syntaxe.

```
# exception ParseErreur ;;
```

La première étape à effectuer est de retrouver les opérateurs à partir des symboles :

```
# let symb_unr = function
  "!" -> NON | "-" -> OPPOSE | _ -> raise ParseErreur
let symb_bin = function
  "+" -> PLUS | "-" -> MOINS | "*" -> MULT | "/" -> DIV |
  | "=" -> EGAL | "<" -> INF | "<=" -> INFEQ | ">" -> SUP |
  | ">=" -> SUPEQ | "<>" -> DIFF | "&" -> ET | "|" -> OU |
  | _ -> raise ParseErreur
```

```

let tsymb s = try Tbin (symb_bin s) with ParseErreur ->
val symb_unr : string -> op_unr = <fun>
val symb_bin : string -> op_bin = <fun>
val tsymb : string -> exp_elem = <fun>

```

La fonction suivante (reduit) implante l'opération de réduction de la pile. Il y a deux cas à considérer, la pile commence par :

- une expression suivie d'un opérateur unaire
- une expression suivie d'un opérateur binaire et d'une expression.

De plus, réduit prend en argument la priorité minimale qu'un opérateur doit avoir pour que la réduction de la pile ait lieu. Pour réduire sans condition, il suffit de donner une profondeur minimale nulle.

```

# let réduit pr = function
  (Texp e)::(Tunr op)::st when (priority_ou op) >= pr
  -> (Texp (ExpUnr (op,e)))::st
  | (Texp e1)::(Tbin op)::(Texp e2)::st when (priority_ou op) >= pr
  -> (Texp (ExpBin (e2,op,e1)))::st
  | _ -> raise ParseErreur ;;
val réduit : int -> exp_elem list -> exp_elem list = <fun>

```

Notez que les éléments d'expressions sont empilés suivant l'ordre de lecture, d'où la nécessité d'inverser les deux opérandes d'une opération binaire.

La fonction principale de l'analyse syntaxique est la fonction empile_ou_reduit qui, suivant le lexème donné en argument, ajoute un nouvel élément dans la pile ou procède à une réduction.

```

# let rec empile_ou_reduit lex stack = match lex , stack
  | Lint n , _ -> (Texp (ExpInt n))::stack
  | Lident v , _ -> (Texp (ExpVar v))::stack
  | Lstring s , _ -> (Texp (ExpStr s))::stack
  | Lsymbol "(" , _ -> Tpg::stack
  | Lsymbol ")" , (Texp e)::Tpg::st -> (Texp e)::st
  | Lsymbol ")" , _ -> empile_ou_reduit lex (réduit 0 stack)
  | Lsymbol s , _

```



```

-> let symbole =
if s<>"-" then tsymb s
(* lever l'ambiguïté du symbole ``-'') *)
(* suivant la pile (i.e dernier exp_elem empilé) *)
else match stack
with (Texp _)::_ -> Tbin MOINS
| _ -> Tunr OPPOSE
in ( match symbole with
Tunr op -> (Tunr op)::stack
| Tbin op ->
( try empile_ou_reduit lex (reduit (priority_ob op)
stack )
with ParseErreur -> (Tbin op)::stack )
| _ -> raise ParseErreur )
| _ , _ -> raise ParseErreur ;;
val empile_ou_reduit : lexeme -> exp_elem list -> exp_el

```

Une fois tous les lexèmes isolés et empilés, l'arbre de syntaxe abstraite doit être finalement construit avec les éléments restant dans la pile. C'est le but de la fonction `reduit_tout`. Si l'expression à analyser est bien formée, il ne doit plus rester dans la pile qu'un seul élément contenant l'arbre de cette expression.

```

# let rec reduit_tout = function
| [] -> raise ParseErreur
| [Texp x] -> x
| st -> reduit_tout (reduit 0 st) ;;
val reduit_tout : exp_elem list -> expression = <fun>

```

La fonction `parse_exp` est la fonction principale de l'analyse des expressions. Elle parcourt une chaîne de caractères, en extrait les différents lexèmes et les passe à la fonction `empile_ou_reduit`. L'analyse s'arrête lorsque le lexème courant satisfait un prédicat passé en argument.

```

# let parse_exp fin cl =
let p = ref 0 in
let rec parse_un stack =
let l = ( p:=cl.courant ; lexer cl)
in if not (fin l) then parse_un (empile_ou_reduit l sta
else ( cl.courant <- !p ; reduit_tout stack )

```

```

in parse_un [] ;;
val parse_exp : (lexeme -> bool) -> chaine_lexer -> expr

```

Notons que le lexème qui a déterminé l'arrêt du parcours n'a pas servi à construire l'expression. Il faut donc restaurer son indice de début (variable p) pour qu'il puisse être traité par la suite.

Passons maintenant à l'analyse d'une ligne d'instruction :

```

# let parse_inst cl = match lexer cl with
  Lident s -> ( match s with
    "REM" -> Rem (extrait (fun _ -> true) cl)
  | "GOTO" -> Goto (match lexer cl with
    Lint p -> p
  | _ -> raise ParseErreur)
  | "INPUT" -> Input (match lexer cl with
    Lident v -> v
  | _ -> raise ParseErreur)
  | "PRINT" -> Print (parse_exp (|=) Lfin) cl)
  | "LET" ->
let l2 = lexer cl and l3 = lexer cl
in ( match l2 ,l3 with
  (Lident v,Lsymbol "=") -> Let (v,parse_exp (|=) Lfin) cl
  | _ -> raise ParseErreur )
  | "IF" ->
let test = parse_exp (|=) (Lident "THEN")) cl
in ( match ignore (lexer cl) ; lexer cl with
  Lint n -> If (test,n)
  | _ -> raise ParseErreur )
  | _ -> raise ParseErreur )
  | _ -> raise ParseErreur ;;
val parse_inst : chaine_lexer -> instruction = <fun>

```

Enfin, voici la fonction principale d'analyse syntaxique des commandes entrées par l'utilisateur depuis la boucle d'interaction :

```

# let parse str =
  let cl = init_lex str
  in match lexer cl with
    Lint n -> Ligne { num=n ; inst=parse_inst cl }

```

```

| Lident "LIST" -> List
| Lident "RUN" -> Run
| Lident "END" -> End
| _ -> raise ParseErreur ;;
val parse : string -> phrase = <fun>

```

Évaluation

Un programme Basic est constitué d'une suite de lignes d'instruction. Son exécution démarre à la première ligne. L'interprétation d'une ligne de programme consiste donc à effectuer le travail demandé par l'instruction qui s'y trouve. Il y a trois familles d'instruction : les entrées-sorties (**PRINT** et **INPUT**), les déclarations ou modifications de variables (**LET**) et les branchements (**GOTO** et **THEN**). Les instructions d'entrées-sorties gèrent l'interaction avec l'utilisateur et exécuteront les fonctions équivalentes en Objective CAML.

Les déclarations de variables et leurs modifications ont besoin de savoir calculer la valeur d'une expression arithmétique et de connaître la localisation mémoire de la variable. L'évaluation d'une expression retourne une valeur entière ou booléenne ou des chaînes de caractères. Nous les regroupons sous le type valeur.

```
# type valeur = Vint of int | Vstr of string | Vbool of
```

La déclaration d'une variable doit pouvoir réserver une place mémoire pour le rangement de la valeur qui lui est attribuée. De même la modification d'une variable nécessite la modification de la valeur associée à son nom. Pour cela l'évaluation d'un programme Basic utilise un *environnement* qui contient la liaison entre un nom de variable et sa valeur. On le représente par une liste d'associations, liste de couples (nom,valeur) :

```
# type environnement = (string * valeur) list ;;
```

On accède au contenu d'une variable par son nom. La modification de la valeur d'une variable change l'association.

Les instructions de branchements, inconditionnels ou conditionnels, précisent à quel numéro de ligne l'exécution du programme doit se poursuivre. Par défaut, la nouvelle instruction à exécuter est la suivante. Pour cela il est nécessaire de conserver le numéro de la prochaine ligne à exécuter.

La structure de liste d'instructions qui représente le programme édité sous la boucle d'interaction ne convient pas à une exécution raisonnablement efficace du programme. En effet, pour réaliser les instructions de saut (`If` et `Goto`) nous devrions reparcourir la totalité de la liste d'instructions pour trouver la ligne indiquée par le saut. Il est possible d'accéder directement à une ligne indiquée par une instruction de saut en substituant à la structure de liste, une structure de tableau et en utilisant les indices de localisation des instructions à la place des numéros de ligne. Pour cela, nous soumettrons l'ensemble des instructions du programme à un prétraitement, dit d'assemblage avant chaque exécution réclamée par une commande `RUN`. Pour des raisons de commodité que nous expliquons au paragraphe suivant, nous représentons un programme assemblé pour l'exécution non pas comme un simple tableau d'instructions, mais comme un tableau de lignes :

```
# type code = ligne array ;;
```

Comme pour la calculatrice des chapitres précédents, l'évaluateur manipule un état que l'évaluation de chaque instruction modifie. Les informations à connaître à chaque instant sont le programme en entier, la prochaine ligne à exécuter et les valeurs des variables. Le programme exécuté n'est pas exactement celui construit sous la boucle d'interaction : plutôt que d'exécuter une liste d'instructions, nous manipulerons un tableau d'instructions. L'état d'exécution d'un programme est donc :

```
# type etat_exec = { ligne:int ; xprog:code ; xenv:envir
```

Il y a deux sources d'erreur à l'évaluation d'une ligne de programme : le calcul d'une expression et un branchement sur une ligne qui n'existe pas. Il nous faut donc les gérer pour que l'interprète s'arrête

correctement en affichant un message d'erreur. On définit une exception et une fonction la déclenchant tout en indiquant à quel numéro de ligne elle s'est produite.

```
# exception RunErreur of int
  let runerr n = raise (RunErreur n) ;;
exception RunErreur of int
val runerr : int -> 'a = <fun>
```

Assemblage

Le processus d'assemblage d'un programme donné sous forme de liste de lignes numérotées (type *program*) consiste à transformer cette liste en tableau puis à ajuster les instructions de saut. Pour réaliser cet ajustement, il suffit de conserver l'association entre les numéros de lignes et l'indice correspondant du tableau. C'est pour obtenir facilement cette association que nous construisons un tableau de lignes numérotées. Pour retrouver l'indice auquel est associé une ligne, on parcourt ce tableau. Si le numéro de ligne recherché n'est pas trouvé, on lui assigne, par convention, l'indice erroné -1.

```
# exception Resultat_cherche_indice of int ;;
exception Resultat_cherche_indice of int
# let cherche_indice tprog num_ligne =
  try
    for i=0 to (Array.length tprog)-1 do
      let mun_i = tprog.(i).num
      in if mun_i=num_ligne then raise (Resultat_cherche_indice num_ligne)
      else if mun_i>num_ligne then raise (Resultat_cherche_indice num_ligne)
    done ;
    (-1 )
  with Resultat_cherche_indice i -> i ;;
val cherche_indice : ligne array -> int -> int = <fun>

# let assemble prog =
  let tprog = Array.of_list prog in
  for i=0 to (Array.length tprog)-1 do
    match tprog.(i).inst with
```

```

Goto n -> let indice = cherche_indice tprog n
in tprog.(i) <- { tprog.(i) with inst = Goto indice }
| If(c,n) -> let indice = cherche_indice tprog n
in tprog.(i) <- { tprog.(i) with inst = If (c,indice) }
| _ -> ()
done ;
tprog ;;
val assemble : ligne list -> ligne array = <fun>

```

Évaluation des expressions

La fonction d'évaluation des expressions parcourt en profondeur l'arbre de syntaxe abstraite et effectue les opérations indiquées à chaque noeud de l'arbre.

L'exception RunErreur est déclenchée dans les cas suivants :
incohérence de type, division par zéro, variable non déclarée.

```

# let rec eval_exp n envt expr = match expr with
  ExpInt p -> Vint p
  | ExpVar v -> ( try List.assoc v envt with Not_found ->
  | ExpUnr (OPPOSE,e) ->
    ( match eval_exp n envt e with
  Vint p -> Vint (-p)
  | _ -> runerr n )
  | ExpUnr (NON,e) ->
    ( match eval_exp n envt e with
  Vbool p -> Vbool (not p)
  | _ -> runerr n )
  | ExpStr s -> Vstr s
  | ExpBin (e1,op,e2)
-> match eval_exp n envt e1 , op , eval_exp n envt e2 with
  Vint v1 , PLUS , Vint v2 -> Vint (v1 + v2)
  | Vint v1 , MOINS , Vint v2 -> Vint (v1 - v2)
  | Vint v1 , MULT , Vint v2 -> Vint (v1 * v2)
  | Vint v1 , DIV , Vint v2 when v2<>0 -> Vint (v1 / v2)
  | Vint v1 , MOD , Vint v2 when v2<>0 -> Vint (v1 mod v2)
  | Vint v1 , EGAL , Vint v2 -> Vbool (v1 = v2)

```

```

| Vint v1 , DIFF , Vint v2  ->  Vbool (v1 <> v2)
| Vint v1 ,  INF  , Vint v2  ->  Vbool (v1 < v2)
| Vint v1 ,  SUP  , Vint v2  ->  Vbool (v1 > v2)
| Vint v1 , INFEQ , Vint v2  ->  Vbool (v1 <= v2)
| Vint v1 , SUPEQ , Vint v2  ->  Vbool (v1 >= v2)

| Vbool v1 , ET , Vbool v2  ->  Vbool (v1 && v2)
| Vbool v1 , OU , Vbool v2  ->  Vbool (v1 || v2)

| Vstr v1 , PLUS , Vstr v2 -> Vstr (v1 ^ v2)
| _ , _ , _ -> runerr n ;;
val eval_exp : int -> (string * valeur) list -> expression

```

Évaluation des instructions

Pour réaliser la fonction d'évaluation d'une ligne d'instruction, nous avons besoin de quelques fonctions auxiliaires.

On ajoute une nouvelle association dans un environnement en supprimant éventuellement la liaison d'une variable de même nom :

```

# let rec ajoute v e env = match env with
  [] -> [v,e]
  | (w,f)::l -> if w=v then (v,e)::l else (w,f)::(ajoute
val ajoute : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list

```

Enfin l'affichage d'une valeur d'un entier ou d'une chaîne de caractères sera utile pour l'instruction PRINT.

```

# let print_valeur v = match v with
  Vint n -> print_int n
  | Vbool true -> print_string "true"
  | Vbool false -> print_string "false"
  | Vstr s -> print_string s ;;
val print_valeur : valeur -> unit = <fun>

```

L'évaluation d'une instruction est une transition menant d'un état à un autre. En particulier sont modifiés d'une part l'environnement si l'instruction est une affectation et d'autre part la prochaine ligne à exécuter. Par convention, si la prochaine ligne à exécuter sort du

tableau contenant le code du programme, sa valeur est -1.

```
# let ligne_suivante etat =  
  let n = etat.ligne+1 in  
  if n < Array.length etat.xprog then n else -1 ;;  
val ligne_suivante : etat_exec -> int = <fun>  
# let eval_inst etat =  
  match etat.xprog.(etat.ligne).inst with  
  Rem _    -> { etat with ligne = ligne_suivante etat }  
  | Print e -> print_valeur (eval_exp etat.ligne etat.x  
  print_newline () ;  
  { etat with ligne = ligne_suivante etat }  
  | Let(v,e) -> let ev = eval_exp etat.ligne etat.xenv e  
  in { etat with ligne = ligne_suivante etat ;  
  xenv = ajoute v ev etat.xenv }  
  | Goto n    -> { etat with ligne = n }  
  | Input v   -> let x = try read_int ()  
  with Failure "int_of_string" -> 0  
  in { etat with ligne = ligne_suivante etat;  
  xenv = ajoute v (Vint x) etat.xenv }  
  | If (t,n) -> match eval_exp etat.ligne etat.xenv t with  
  Vbool true -> { etat with ligne = n }  
  | Vbool false -> { etat with ligne = ligne_suivante etat  
  | _ -> runerr etat.ligne ;;  
val eval_inst : etat_exec -> etat_exec = <fun>
```

À chaque appel, la fonction de transition eval_inst cherche la ligne à exécuter (variable lc) et le numéro de la ligne suivante (variable ns). Si la dernière ligne du programme est atteinte, on attribue à ns la valeur -1. Cela nous permettra d'arrêter l'exécution.

Évaluation d'un programme

On applique récursivement les transitions jusqu'à obtenir un état où le numéro de la ligne courante est -1.

```
# let rec run etat =  
  if etat.ligne = -1 then etat else run (eval_inst etat)  
val run : etat_exec -> etat_exec = <fun>
```


Mise en oeuvre

Il ne nous reste plus qu'à réaliser un mini-éditeur et d'assembler toutes les briques réalisées dans les sections précédentes.

La fonction `inserer` ajoute, à la bonne place, une nouvelle ligne dans un programme.

```
# let rec inserer ligne p = match p with
| [] -> [ligne]
| l::prog ->
  if l.num < ligne.num then l::(inserer ligne prog)
  else if l.num=ligne.num then ligne::prog
  else ligne::l::prog ;;
val inserer : ligne -> ligne list -> ligne list = <fun>
```

La fonction `print_prog` affiche le source d'un programme.

```
# let print_prog prog =
  let print_ligne x = print_string (pp_ligne x) ; print_r
  print_newline () ;
  List.iter print_ligne prog ;
  print_newline () ;;
val print_prog : ligne list -> unit = <fun>
```

La fonction `une_commande` traite l'ajout d'une ligne ou le lancement d'une commande. Elle agit sur l'état de la boucle d'interaction constitué d'un programme et d'un environnement. Cet état, représenté par le type *etat_boucle*, n'est pas celui de l'évaluation d'un programme.

```
# type etat_boucle = { prog:program; env:environnement }
# exception Fin ;;
```

```
# let une_commande etat =
```

```

print_string "> " ; flush stdout ;
try
match parse (input_line stdin) with
  Ligne l -> { etat with prog = inserer l etat.prog }
  | List -> (print_prog etat.prog ; etat )
  | Run
-> let tprog = assemble etat.prog in
let xetat = run { ligne = 0; xprog = tprog; xenv = etat
{etat with env = xetat.xenv }
  | End -> raise Fin
with
  LexerErreur -> print_string "Illegal character\n"; etat
  | ParseErreur -> print_string "syntax error\n"; etat
  | RunErreur n ->
print_string "runtime error at line ";
print_int n ;
print_string "\n";
etat ;;
val une_commande : etat_boucle -> etat_boucle = <fun>

```

La fonction principale est la fonction go qui lance la boucle d'interaction de notre Basic.

```

# let go () =
  try
    print_string "Mini-BASIC version 0.1\n\n";
    let rec loop etat = loop (une_commande etat) in
    loop { prog = []; env = [] }
    with Fin -> print_string "A bientôt...\n";;
val go : unit -> unit = <fun>

```

La boucle est réalisée par la fonction locale loop. On en sort lorsque l'exception Fin a été déclenchée par la fonction une_commande.

Exemple : C+ /C-

On reprend l'exemple du jeu C+ /C- décrit au chapitre 3, page ??.
Voici le programme Basic équivalent au programme Objective CAML donné :

```
10 PRINT "Donner le nombre cache : "  
20 INPUT N  
30 PRINT "Donner un nombre : "  
40 INPUT R  
50 IF R = N THEN 110  
60 IF R < N THEN 90  
70 PRINT "C-"  
80 GOTO 30  
90 PRINT "C+"  
100 GOTO 30  
110 PRINT "BRAVO"
```

Voici, pour finir, la trace de l'exécution de ce programme.

```
> RUN  
Donner le nombre caché :  
64  
Donner un nombre :  
88  
C-  
Donner un nombre :  
44  
C+  
Donner un nombre :  
64  
BRAVO
```

Pour en faire plus

Le Basic que nous venons de programmer reste minimal. Pour ceux qui veulent aller plus loin, nous proposons en exercice quelques pistes pour enrichir l'évaluateur.

1. *Les flottants* : tel quel, notre langage ne connaît que les entiers, les chaînes et les booléens. Ajouter les flottants ainsi que leurs principales opérations dans la grammaire du langage. Outre l'analyse lexicale, il faut modifier l'évaluation en tenant compte des conversions implicites entre entiers et

flottants.

2. *Les tableaux* : il s'agit ici d'ajouter à la syntaxe l'instruction `DIM var[x]` qui permet de déclarer un tableau `var` de taille `x` et l'expression `var[i]` qui référence le *i*ème élément du tableau `var`.
3. *Directives* : nous devrions aussi ajouter les directives `SAVE "nom_fichier"` et `LOAD "nom_fichier"` qui respectivement enregistre ou charge un programme Basic sur ou depuis le disque dur.
4. *Sous-programmes* : l'appel à un sous-programme s'effectue en utilisant l'instruction `GOSUB` numéro de ligne qui effectue un branchement à ce numéro de ligne tout en conservant le numéro de la ligne d'appel. L'instruction `RETURN` reprend l'exécution à la ligne suivant la dernière instruction `GOSUB` exécutée si elle existe et sort du programme sinon. Pour ajouter cette possibilité, il faudra gérer dans l'évaluation, en plus de l'environnement, une pile contenant les adresses de retour des différents appels à `GOSUB`. L'instruction `GOSUB` permet de définir des sous-programmes récursifs.

Démineur

Rappelons brièvement les principes du jeu : explorer un champ de mines sans marcher sur l'une d'elles.

Un champ de mines est un tableau à deux dimensions dont quelques cases contiennent des mines cachées alors que les autres sont vides. En début de partie toutes les cases sont recouvertes et le joueur doit les découvrir l'une après l'autre. Le joueur est victorieux quand il a réussi à découvrir toutes les cases ne contenant pas de mines.

À chaque étape du jeu, le joueur peut au choix ouvrir une case, ou la marquer comme minée. S'il ouvre une case et que celle-ci contient une mine, il saute et perd la partie. Sinon, la case change d'aspect et s'y inscrit le nombre des cases adjacentes qui sont minées (donc au maximum 8). S'il choisit de marquer la case, il ne peut plus ensuite l'ouvrir sans avoir au préalable retiré la marque.



Figure 6.5 : Copie d'écran

Nous décomposons l'implantation de ce jeu en trois parties.

1. La description d'un jeu abstrait comprenant la représentation interne du champ de mines et les fonctions manipulant cette représentation.
2. La description graphique du jeu avec ses fonctions propres pour le rendu de l'affichage des cases.
3. Le moteur gérant les interactions entre le programme et l'utilisateur.

Le champ de mines abstrait

Cette partie ne s'intéresse qu'au champ de mines considéré comme

une pure abstraction, nous ne nous occupons pas de son affichage.

Configuration

Un champ de mines est caractérisé par ses dimensions et le nombre de mines qu'il contient. Nous regroupons ces trois paramètres dans un enregistrement et définissons une configuration par défaut : 10×10 cases et 15 mines.

```
# type config = {  
  nbcols  : int ;  
  nbrows  : int ;  
  nbmines : int };;  
# let default_config = { nbcols=10; nbrows=10; nbmines=15 }
```

Le champ de mines

Il apparaît naturel qu'un champ de mines soit codé par un tableau à deux dimensions. Il nous faut encore préciser ce que sont ces cases en déterminant les informations qu'il est nécessaire de connaître pour chacune d'elle. L'état d'une case peut être :

- la case est-elle minée ?
- la case a-t-elle été découverte ?
- la case a-t-elle été marquée ?
- combien y a-t-il de cases minées parmi les cases adjacentes ?

Cette dernière information n'est pas indispensable puisqu'il est possible de la calculer au moment opportun, mais il est plus simple de faire ce calcul une fois pour toutes en début de partie.

Nous représentons une case par un enregistrement contenant ces quatre informations.

```
# type cell = {  
  mutable mined : bool ;  
  mutable seen  : bool ;  
  mutable flag  : bool ;
```

```
mutable nbm : int
} ;;
```

Le tableau à deux dimensions du champ de mine est un tableau de tableaux de cases :

```
# type board = cell array array ;;
```

Un itérateur

Dans la suite du programme nous aurons à de multiples occasions besoin d'appliquer une fonction à chacune des cases du champ de mines. Pour le faire génériquement, nous définissons l'itérateur `iter_on_cell` appliquant la fonction `f` donnée en argument à chaque case du tableau défini par la configuration `cf`.

```
# let iter_on_cell cf f =
  for i=0 to cf.nbcolls-1 do for j=0 to cf.nbrows-1 do f (
val iter_on_cell : config -> (int * int -> 'a) -> unit =
```

Nous avons ici un bon exemple du mélange des styles fonctionnel et impératif, puisque pour itérer une fonction opérant par effet de bord (puisque sans résultat) sur toutes les coordonnées d'un tableau nous utilisons une fonctionnelle (une fonction prenant en argument une autre fonction).

Initialisation

Nous déterminons aléatoirement l'ensemble des cases minées. Si l et c sont respectivement le nombre de lignes et le nombre de colonnes du champ de mines et m le nombre de mines, il faut engendrer une liste de m nombres différents compris entre 1 et $l \times c$. On supposera que $m \leq l \times c$ pour définir l'algorithme, mais le programme devra explicitement vérifier cette condition.

L'idée la plus naturelle et la plus simple est de commencer avec une liste vide, de tirer un nombre au hasard, de l'introduire dans la liste

s'il n'y apparaît pas déjà; et de recommencer jusqu'à ce que la liste contienne m nombres. Pour cela nous utiliserons les fonctions suivantes prises dans les modules Random et Sys :

- `Random.int : int -> int` qui tire selon un générateur de nombres aléatoires un nombre compris entre 0 et $n-1$ si n est l'argument passé ;
- `Random.init : int -> unit`, qui initialise le générateur de nombres aléatoires ;
- `Sys.time : unit -> float`, qui retourne le nombre de millisecondes de temps processeur utilisé depuis le début de l'exécution du programme. Cette fonction sera utilisée pour initialiser le générateur de nombres aléatoires de manière différente à chaque partie.

Les modules auxquels appartiennent ces fonctions sont détaillés au chapitre 8, pages ?? et ??.

La fonction de tirage aléatoire des mines prend en argument le nombre de cases (lc) ainsi que le nombre de mines à tirer (m) et retourne une liste de positions linéarisées des m mines.

```
# let random_list_mines lc m =  
  let cell_list = ref []  
  in while (List.length !cell_list) < m do  
    let n = Random.int lc in  
    if not (List.mem n !cell_list) then cell_list := n :: !  
  done ;  
  !cell_list ;;  
val random_list_mines : int -> int -> int list = <fun>
```

Telle qu'elle est écrite, nous ne sommes pas capable de dire en combien d'étapes cette fonction se termine. Si notre générateur aléatoire est fiable, nous pouvons uniquement assurer que la probabilité qu'elle ne termine pas est nulle. Ce qui nous donne la proposition paradoxale : *< < la fonction termine si elle tourne infiniment > >*.

Cependant, la pratique n'a jamais mis en défaut notre fonction. Aussi nous contenterons nous de cette définition incertaine pour engendrer la liste des cases minées.

Il nous faut initialiser le générateur de nombres aléatoires pour que chaque exécution du jeu ne se fasse pas avec le même champ de mines. On utilise le temps processeur depuis le début de l'exécution du programme pour initialiser le générateur de nombres aléatoires.

```
# let generate_seed () =  
  let t = Sys.time () in  
  let n = int_of_float (t*.1000.0)  
  in Random.init(n mod 100000) ;;  
val generate_seed : unit -> unit = <fun>
```

Dans la pratique on s'aperçoit qu'un même programme prendra souvent le même temps d'exécution ce qui entraîne un tirage unique de la fonction `generate_seed`. On utilisera de préférence la fonction `Unix.time` (voir chapitre 18).

Aussi bien lors de l'initialisation du champ de mines que lors du jeu, nous aurons besoin de connaître l'ensemble des cases adjacentes à une case donnée (fonction `neighbours`). L'énumération de cet ensemble doit prendre en compte les cases des bords qui ont moins de voisins que celles du milieu du jeu (fonction `valid`).

```
# let valid cf (i,j) = i>=0 && i<cf.nbcols && j>=0 && j<cf.nbrs  
val valid : config -> int * int -> bool = <fun>  
# let neighbours cf (x,y) =  
  let ngb = [x-1,y-1; x-1,y; x-1,y+1; x,y-1; x,y+1; x+1,y-1; x+1,y; x+1,y+1]  
  in List.filter (valid cf) ngb ;;  
val neighbours : config -> int * int -> (int * int) list
```

La fonction `initialize_board` crée le champ de mines initial. Elle procède en quatre étapes :

1. génération de la liste des cases minées;
2. création d'un tableau à deux dimensions contenant des cellules différentes;
3. marquage des cases minées;
4. décompte des cases minées adjacentes à chaque case non minée.

La fonction `initialize_board` utilise un certain nombre de fonctions locales que nous décrivons brièvement.

`cell_init`

: crée une valeur initiale ;

`copy_cell_init`

: affecte une case du tableau avec une nouvelle copie de la valeur initiale ;

`set_mined`

: dépose une mine dans une case ;

`count_mined_adj`

: compte le nombre de cases minées adjacentes à une case donnée ;

`set_count`

met à jour le nombre de cases minées adjacentes à une case non minée.

```
# let initialize_board cf =
  let cell_init () = { mined=false; seen=false; flag=false }
  let copy_cell_init b (i,j) = b.(i).(j) <- cell_init() i
  let set_mined b n = b.(n / cf.nbrows).(n mod cf.nbrows)
  in
  let count_mined_adj b (i,j) =
    let x = ref 0 in
    let inc_if_mined (i,j) = if b.(i).(j).mined then incr x
    in List.iter inc_if_mined (neighbours cf (i,j)) ;
    !x
  in
  let set_count b (i,j) =
    if not b.(i).(j).mined
    then b.(i).(j).nbm <- count_mined_adj b (i,j)
  in
  let list_mined = random_list_mines (cf.nbcols*cf.nbrows)
  let board = Array.make_matrix cf.nbcols cf.nbrows (cell_init)
  in iter_on_cell cf (copy_cell_init board) ;
  List.iter (set_mined board) list_mined ;
  iter_on_cell cf (set_count board) ;
  board ;;
val initialize_board : config -> cell array array = <fun
```

Découverte

Durant une partie, lorsqu'un joueur découvre une case dont aucun voisin n'est miné, il sait qu'il peut découvrir toutes les cases adjacentes sans risque, et ainsi de suite tant qu'il découvre d'autres cases sans voisin miné. Afin de soulager le joueur de cette étape fastidieuse (phase du jeu n'entraînant pas la moindre réflexion), notre Démineur découvre lui-même toutes ces cases. Pour ce faire, nous écrivons la fonction `cells_to_see` qui renvoie la liste de toutes les cases à découvrir quand une case est ouverte.

L'algorithme est simple à exprimer : si la case découverte a des voisins minés alors la liste se résume à cette unique case; sinon, c'est l'ensemble de ses voisins plus les listes des cases à découvrir quand on découvre chacun des voisins. La difficulté est d'écrire un programme qui ne boucle pas car bien évidemment une case est un voisin de son voisin. Il faut donc faire en sorte de ne pas examiner plusieurs fois les mêmes cases.

Pour garder une trace des cases examinées, on utilise le tableau de booléens `visited`. Il est de même dimension que le champ de mines, on y marque avec la valeur **true** qu'une case a déjà été examinée. On applique récursivement la recherche des cases à découvrir uniquement sur les cases non marquées.

On utilise la fonction auxiliaire `relevant` qui calcule, à partir de la liste des voisins d'une case deux sous-listes. Chacune de ces sous-listes ne contient que des voisins ni minés, ni déjà découverts, ni marqués par le joueur et ni déjà visités. La première sous-liste contient les voisins ayant au moins un voisin miné ; la seconde contient les voisins n'ayant pas de voisin miné. On en profite pour marquer comme visitées toutes ces cases. Remarquons que nous excluons les cases marquées par le joueur même si elles ne sont pas minées, car le sens de cette marque est d'interdire la découverte de la case.

La fonction locale `cells_to_see_rec` réalise la boucle récursive de recherche. Elle construit la liste des cases à découvrir et prend en argument la liste des cases à examiner qui est maintenue à jour. On démarre avec la liste contenant uniquement la dernière case

découverte après l'avoir marquée comme visitée.

```
# let cells_to_see bd cf (i,j) =  
  let visited = Array.make_matrix cf.nbcolls cf.nbrows false  
  let rec relevant = function  
    [] -> ([],[])  
    | ((x,y) as c)::l ->  
      let cell=bd.(x).(y)  
      in if cell.mined || cell.flag || cell.seen || visited.(x).(y)  
      then relevant l  
      else let (l1,l2) = relevant l  
      in visited.(x).(y) <- true ;  
      if cell.nbm=0 then (l1,c::l2) else (c::l1,l2)  
      in  
      let rec cells_to_see_rec = function  
        [] -> []  
        | ((x,y) as c)::l ->  
          if bd.(x).(y).nbm<>0 then c :: (cells_to_see_rec l)  
          else let (l1,l2) = relevant (neighbours cf c)  
          in (c :: l1) @ (cells_to_see_rec (l2 @ l))  
          in visited.(i).(j) <- true ;  
          cells_to_see_rec [(i,j)] ;;  
val cells_to_see :  
  cell array array -> config -> int * int -> (int * int)
```

À première vue, l'argument de `cells_to_see_rec` peut croître entre deux appels alors que la récurrence se fait sur cet argument. On peut donc se demander si cette fonction termine toujours ?

L'utilisation du tableau `visited` nous garantit qu'une case déjà marquée ne peut être dans le résultat de `relevant`. Or les cases venant augmenter la liste des cases à examiner proviennent toutes de `relevant`. De plus, cette fonction marque toutes les cases qu'elle renvoie. Ceci nous garantit qu'une case ne pourra être renvoyée qu'une seule fois par `relevant` donc n'être qu'une seule fois dans la liste des cases à examiner. Le nombre de cases possibles étant fini, il en découle que la fonction termine.

Ceci clôt la partie hors affichage du Démineur, profitons-en pour faire un point sur le style de programmation que nous avons adopté. Le choix de structures modifiables (tableaux et champs mutables)

nous a imposé un style impératif utilisant des boucles et des affectations. Pourtant, pour traiter les problèmes auxiliaires nous avons fait usage de listes et les fonctions de traitement adoptent un style fonctionnel. En fait, le style de programmation nous a été dicté par les structures de données que nous souhaitions manipuler. Le cas de la fonction `cells_to_seen` est symptomatique de cela : c'est une fonction manipulant des listes, et il est apparu naturel d'adopter un style fonctionnel. Pourtant, nous utilisons un tableau pour identifier les cases déjà traitées et nous dérogeons à la fonctionnalité pour maintenir ce tableau. Nous aurions pu dans un pur style fonctionnel maintenir une liste des cases traitées et vérifier si une case apparaissait ou non dans cette liste. Mais le coût de cette méthode aurait été important (la recherche de la présence d'un élément est linéaire suivant la taille de la liste alors qu'avec un tableau elle est constante), et elle ne simplifiait pas le programme.

Affichage du jeu Démineur

Cette partie est dépendante des structures de données représentant l'état du jeu (voir page ??). Il s'agit principalement d'afficher les différents composants de la fenêtre du Démineur suivant la figure 6.6. On utilise pour cet affichage les fonctions sur les boîtes vues page ??.

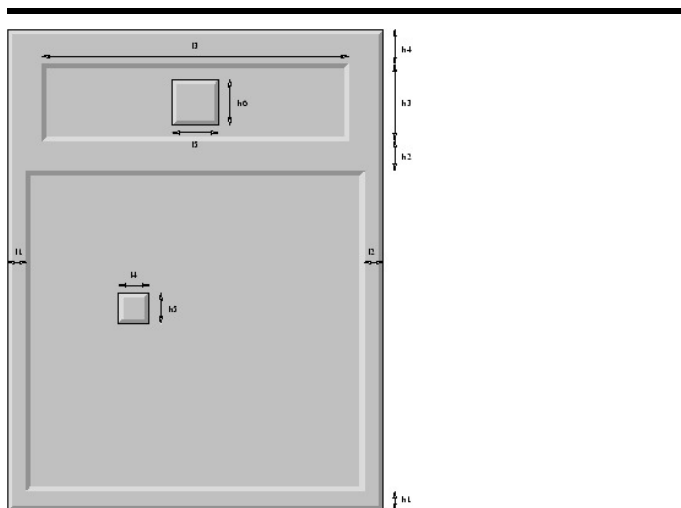


Figure 6.6 : La fenêtre principale du Démineur

Les paramètres suivants définissent les caractéristiques des différents composants de la fenêtre graphique.

```
# let b0 = 31;;
# let h2 = 30 ;;
# let h3 = 15+20 + 2*b0 ;;
# let h4 = 20 + 2*b0 ;;
# let h5 = 20+2*abs0+2*config.nbcols + 2*b0 ;;
# let h5 = 40 + 2*b0 ;;
```

Nous les utilisons pour étendre la configuration de base du démineur (valeur de type *config*). Nous définissons l'enregistrement *window_config* ci-dessous. Le champ *cf* contient la configuration minimale. On associe une boîte à chacun des composants de l'affichage : fenêtre globale (champ *g_bcf*), champ de mines (champ *f_bcf*), fenêtre de dialogue (champ *m_bcf*) avec deux sous boîtes (champs *m1_bcf* et *m2_bcf*), bouton de marquage (champ *s_bcf*), case courante (champ *cc_bcf*).

```
# type window_config = {
  cf : config ;
  g_bcf : box_config ;
  f_bcf : box_config ;
  m_bcf : box_config ;
  m1_bcf : box_config ;
  m2_bcf : box_config ;
  s_bcf : box_config ;
  mutable cc_bcf : box_config ;
  cell : int*int -> (int*int) ;
  coor : int*int -> (int*int)
} ;;
```

De plus, une valeur de type *window_config* fournit deux fonctions :

- *cell* : qui prend les coordonnées d'une case et rend les coordonnées de la boîte correspondante.

- `coor` : qui prend les coordonnées d'un pixel de la fenêtre et rend les coordonnées de la case correspondante.

Configuration

On définit ensuite une fonction construisant une configuration graphique (de type *window_config*) en fonction d'une configuration minimale (de type *config*) et des paramètres définis plus haut. Les valeurs des paramètres de certains composants dépendent les uns des autres. Par exemple, la largeur de la boîte globale dépend de la largeur du champ de mines qui dépend à son tour du nombre de colonnes. Pour ne pas recalculer plusieurs fois une même valeur, nous renseignons incrémentalement les champs des composants. Cette phase d'initialisation d'un ensemble graphique est toujours un peu fastidieuse en l'absence de primitives ou d'outils adéquats.

```
# let make_box x y w h bw r =
  { x=x; y=y; w=w; h=h; bw=bw; r=r; b1_col=grey1; b2_col=
val make_box : int -> int -> int -> int -> int -> relief
  <fun>
# let make_wcf cf =
  let wcols = b0 + cf.nbcols*14 + b0
  and hrows = b0 + cf.nbrows*h5 + b0 in
  let g_bcf = let gw = (b0 + l1 + wcols + l2 + b0)
  and gh = (b0 + h1 + hrows + h2 + h3 + h4 + b0)
  in make_box 0 0 gw gh b0 Top
  and f_bcf = make_box l1 h1 wcols hrows b0 Bot in
  let m_bcf = make_box ((g_bcf.w - 13) / 2) (b0+h1+hrows
  let m1_bcf = make_box (m_bcf.x + b0) (b0 + h1 + hrows +
  ((13-15)/2-(2*b0)) (h3-(2*b0)) 5 Flat in
  let s_bcf = make_box (m1_bcf.x + m1_bcf.w)
  (m1_bcf.y + (h3-h6) / 2) 15 h6 b0 Top in
  let m2_bcf = make_box (s_bcf.x + s_bcf.w)
  m1_bcf.y m1_bcf.w m1_bcf.h 5 Flat in
  let cc_bcf = make_box 0 0 14 h5 b0 Top
  in { cf = cf;
  g_bcf = g_bcf; f_bcf=f_bcf; m_bcf=m_bcf; m1_bcf=m1_bcf;
  s_bcf=s_bcf; m2_bcf=m2_bcf; cc_bcf = cc_bcf;
  cell = (fun (i,j) -> ( l1+b0+14*i , h1+b0+h5*j)) ;
```

```

    coor = (fun (x,y) -> ( (x-11)/14 , (y-11)/h5 )) } ;;
val make_wcf : config -> window_config = <fun>

```

Affichages des cases

Il s'agit maintenant de définir les fonctions réalisant l'affichage des cases dans les différents cas possibles. Une case pourra être ouverte ou fermée et pourra contenir ou non une information. On affichera toujours (la boîte de) la cellule courante de la configuration du jeu (champs `cc_bcf`).

Nous définissons donc deux fonctions modifiant la configuration de la cellule courante ; l'une la fermant, l'autre l'ouvrant.

```

# let close_ccell wcf i j =
  let x,y = wcf.cell (i,j)
  in wcf.cc_bcf <- {wcf.cc_bcf with x=x; y=y; r=Top} ;;
val close_ccell : window_config -> int -> int -> unit =
# let open_ccell wcf i j =
  let x,y = wcf.cell (i,j)
  in wcf.cc_bcf <- {wcf.cc_bcf with x=x; y=y; r=Flat} ;;
val open_ccell : window_config -> int -> int -> unit = <

```

Suivant la phase de jeu, nous serons amenés à afficher une information sur les cases. Nous définissons, pour chaque cas, une fonction particulière.

- Affichage d'une case fermée :

```

# let draw_closed_cc wcf i j =
  close_ccell wcf i j;
  draw_box wcf.cc_bcf ;;
val draw_closed_cc : window_config -> int -> int ->

```

- Affichage d'une case ouverte avec son compte de mines :

```

# let draw_num_cc wcf i j n =
  open_ccell wcf i j ;

```



```

draw_box wcf.cc_bcf ;
if n<>0 then draw_string_in_box Center (string_of_
wcf.cc_bcf Graphics.white ;;
val draw_num_cc : window_config -> int -> int -> in

```

- Affichage d'une case contenant une bombe :

```

# let draw_bomb_cc wcf i j =
  open_ccell wcf i j ;
  let cc = wcf.cc_bcf
  in draw_box wcf.cc_bcf ;
  Graphics.set_color Graphics.black ;
  Graphics.fill_circle (cc.x+cc.w/2) (cc.y+cc.h/2) (
val draw_bomb_cc : window_config -> int -> int -> u

```

- Affichage d'une case minée et marquée :

```

# let draw_flag_cc wcf i j =
  close_ccell wcf i j ;
  draw_box wcf.cc_bcf ;
  draw_string_in_box Center "!" wcf.cc_bcf Graphics.
val draw_flag_cc : window_config -> int -> int -> u

```

- Affichage d'une case marquée à mauvais escient :

```

# let draw_cross_cc wcf i j =
  let x,y = wcf.cell (i,j)
  and w,h = wcf.cc_bcf.w, wcf.cc_bcf.h in
  let a=x+w/4 and b=x+3*w/4
  and c=y+h/4 and d=y+3*h/4
  in Graphics.set_color Graphics.red ;
  Graphics.set_line_width 3 ;
  Graphics.moveto a d ; Graphics.lineto b c ;
  Graphics.moveto a c ; Graphics.lineto b d ;
  Graphics.set_line_width 1 ;;

```

```
val draw_cross_cc : window_config -> int -> int ->
```

En cours de jeu, le choix de la fonction d'affichage d'une case est réalisé par :

```
# let draw_cell wcf bd i j =
  let cell = bd.(i).(j)
  in match (cell.flag, cell.seen , cell.mined ) with
    (true,_,_) -> draw_flag_cc wcf i j
  | (_,false,_) -> draw_closed_cc wcf i j
  | (_,_,true) -> draw_bomb_cc wcf i j
  | _ -> draw_num_cc wcf i j cell.nbm ;;
val draw_cell : window_config -> cell array array -> int
<fun>
```

Une fonction spécifique affichera l'ensemble des cases en fin de partie. Elle diffère légèrement de la précédente car toutes les cases sont considérées comme devant être découvertes. De plus, on signale d'une croix rouge les cases marquées à tort par le joueur.

```
# let draw_cell_end wcf bd i j =
  let cell = bd.(i).(j)
  in match (cell.flag, cell.mined ) with
    (true,true) -> draw_flag_cc wcf i j
  | (true,false) -> draw_num_cc wcf i j cell.nbm; draw_cr
  | (false,true) -> draw_bomb_cc wcf i j
  | (false,false) -> draw_num_cc wcf i j cell.nbm ;;
val draw_cell_end : window_config -> cell array array ->
<fun>
```

Affichage des autres composants

L'état du mode marquage est matérialisé par une boîte en creux ou en relief contenant le mot ON ou OFF :

```
# let draw_flag_switch wcf on =
  if on then wcf.s_bcf.r <- Bot else wcf.s_bcf.r <- Top ;
  draw_box wcf.s_bcf ;
  if on then draw_string_in_box Center "ON" wcf.s_bcf Gra
  else draw_string_in_box Center "OFF" wcf.s_bcf Graphics
```

```
val draw_flag_switch : window_config -> bool -> unit = <
```

Au dessus du bouton de marquage, on affiche son rôle :

```
# let draw_mark_title wcf =  
  let m = "Marquage" in  
  let w,h = Graphics.text_size m in  
  let x = (wcf.g_bcf.w-w)/2  
  and y0 = wcf.m_bcf.y+wcf.m_bcf.h in  
  let y = y0+(wcf.g_bcf.h-(y0+h))/2  
  in Graphics.moveto x y ;  
  Graphics.draw_string m ;;  
val draw_mark_title : window_config -> unit = <fun>
```

Tout au long de la partie, le nombre de cases restant à découvrir et le nombre de cases marquées sont affichés dans la boîte de message, de part et d'autre du bouton de marquage.

```
# let print_score wcf nbcto nbfc =  
  erase_box wcf.m1_bcf ;  
  draw_string_in_box Center (string_of_int nbcto) wcf.m1_bcf ;  
  erase_box wcf.m2_bcf ;  
  draw_string_in_box Center (string_of_int (wcf.cf.nbmines  
    ( if nbfc>wcf.cf.nbmines then Graphics.red else Graphics.black )  
    nbfc)) wcf.m2_bcf ;  
val print_score : window_config -> int -> int -> unit =
```

Pour dessiner le champ de mines initial, il faut dessiner (nombre de lignes) $\times$ (nombre de colonnes) fois une même case fermée. C'est toujours le même dessin, mais il peut être assez long à faire puisqu'il faut dessiner et remplir en plus d'un rectangle, quatre trapèzes. Pour accélérer cette initialisation, nous avons recours à la technique consistant à ne dessiner qu'une seule case, la capturer sous forme d'un bitmap et le reproduire à chaque emplacement d'une case.

```
# let draw_field_initial wcf =  
  draw_closed_cc wcf 0 0 ;  
  let cc = wcf.cc_bcf in  
  let bitmap = draw_box cc ; Graphics.get_image cc.x cc.y cc.x cc.y  
  let draw_bitmap (i,j) = let x,y=wcf.cell (i,j) in  
  in Graphics.draw_image bitmap x y
```

```

in iter_on_cell wcf.cf draw_bitmap ;;
val draw_field_initial : window_config -> unit = <fun>

```

En fin de partie, on découvre la totalité du champ de mines en marquant d'une croix rouge les cases marquées à mauvais escient :

```

# let draw_field_end wcf bd =
  iter_on_cell wcf.cf (fun (i,j) -> draw_cell_end wcf bd)
val draw_field_end : window_config -> cell array array -> unit

```

Enfin, la fonction principale d'affichage de début de partie ouvre le contexte graphique et affiche l'état initial des différents composants.

```

# let open_wcf wcf =
  Graphics.open_graph ( " " ^ (string_of_int wcf.g_bcf.w)
    (string_of_int wcf.g_bcf.h) ) ;
  draw_box wcf.g_bcf ;
  draw_box wcf.m_bcf ;
  draw_flag_switch wcf false ;
  draw_box wcf.f_bcf ;
  draw_field_initial wcf ;
  draw_mark_title wcf ;
  print_score wcf ((wcf.cf.nbrows*wcf.cf.nbcols)-wcf.cf.n_mines)
val open_wcf : window_config -> unit = <fun>

```

Notons que toutes les primitives d'affichage sont paramétrées par une configuration graphique de type *window_config*. Le fait d'avoir procédé de la sorte les rend indépendantes des choix de disposition des éléments composant notre Démineur. Si nous souhaitons modifier cette disposition, le code reste utilisable tel quel, seule la configuration doit être mise à jour.

Interaction entre le programme et le joueur

Définissons les différentes possibilités offertes au joueur :

- il peut cliquer sur la case de sélection pour changer de mode (découverte ou marquage),
- ou cliquer sur l'une des cases pour la découvrir ou la marquer,
- ou il peut appuyer sur la touche 'q' pour quitter le jeu.

Rappelons qu'un événement de Graphics (*Graphics.event*) doit être associé à un enregistrement (*Graphics.status*) contenant les informations courantes sur la souris et le clavier à l'instant où se produit l'événement. Une interaction souris peut avoir lieu soit sur le bouton de marquage, soit sur une case du champ de mines. Tout autre événement souris doit être ignoré. Afin de différencier ces événements souris, nous nous donnons le type :

```
# type clickon = Out | Cell of (int*int) | SelectBox ;;
```

D'autre part, l'appui sur le bouton de la souris et son relâchement sont deux événements distincts. Pour qu'un clic soit valide, nous imposons que les deux événements aient eu lieu sur le même composant (bouton de marquage ou case du champ de mines).

```
# let locate_click wcf st1 st2 =  
  let clickon_of st =  
    let x = st.Graphics.mouse_x and y = st.Graphics.mouse_y  
    in if x>=wcf.s_bcf.x && x<=wcf.s_bcf.x+wcf.s_bcf.w &&  
      y>=wcf.s_bcf.y && y<=wcf.s_bcf.y+wcf.s_bcf.h  
    then SelectBox  
    else let (x2,y2) = wcf.coor (x,y)  
    in if x2>=0 && x2<wcf.cf.nbcols && y2>=0 && y2<wcf.cf.n  
    then Cell (x2,y2) else Out  
  in  
    let r1=clickon_of st1 and r2=clickon_of st2  
    in if r1=r2 then r1 else Out ;;  
val locate_click :  
  window_config -> Graphics.status -> Graphics.status ->
```

Le coeur du programme est la boucle d'attente et de traitement des événements définie dans la fonction loop. Elle s'inspire de la fonction *squel* décrite à la page ??, mais spécifie mieux les types d'événement souris. Cette boucle est interrompue :

- par appui de la touche q (ou Q) que l'on interprète comme la volonté du joueur de terminer la partie ;
- lorsque le joueur découvre une case minée, auquel cas il a perdu ;
- lorsque le joueur a découvert toutes les cases non minées, auquel cas il a gagné.

On rassemble dans le type enregistrement *demin_cf* les différentes informations utiles aux fonctions de traitement de l'interaction :

```
# type demin_cf =  
  { wcf : window_config; bd : cell array array;  
  mutable nb_marked_cells : int;  
  mutable nb_hidden_cells : int;  
  mutable flag_switch_on : bool } ;;
```

Les champs correspondent à :

- wcf : la configuration graphique ;
- bd : le tableau de cellules ;
- flag_switch_on : un booléen indiquant si le jeu est en mode marqué ou non ;
- nb_marked_cells : le nombre de cases marquées ;
- nb_hidden_cells : le nombre de cases non minées restant à découvrir ;

La boucle principale s'écrit de la manière suivante :

```
# let loop d f_init f_key f_mouse f_end =  
  try  
  while true do  
    let st = Graphics.wait_next_event  
      [Graphics.Button_down;Graphics.Key_pressed]  
    in if st.Graphics.keypressed then f_key st.Graphics.key  
    else let st2 = Graphics.wait_next_event [Graphics.Button  
    in f_mouse (locate_click d.wcf st st2)  
    done  
    with Fin -> ();;  
val loop : demin_cf -> 'a -> (char -> 'b) -> (clickon ->  
  <fun>
```

Les fonctions d'initialisation, de fin et de gestion clavier sont fort simples.

```
# let d_init d () = open_wcf d.wcf  
  let d_end () = Graphics.close_graph()  
  let d_key c = if c='q' || c='Q' then raise Fin;;
```

```

val d_init : demin_cf -> unit -> unit = <fun>
val d_end : unit -> unit = <fun>
val d_key : char -> unit = <fun>

```

Par contre la fonction de gestion des événements souris nécessite l'utilisation de quelques fonctions auxiliaires :

- **mark_cell** : réponse à un clic sur une case en mode marquage actif.
- **ending** : fin du jeu. On découvre tout le champ de mines, on affiche un message indiquant victoire ou défaite et on attend un événement souris ou clavier quelconque pour fermer l'application.
- **reveal** : réponse à un clic sur une case en mode découverte (i.e. mode marquage inactif)

```

# let mark_cell d i j =
  if d.bd.(i).(j).flag
  then ( d.nb_marked_cells <- d.nb_marked_cells -1;
    d.bd.(i).(j).flag <- false )
  else ( d.nb_marked_cells <- d.nb_marked_cells +1 ;
    d.bd.(i).(j).flag <- true ) ;
  draw_cell d.wcf d.bd i j ;
  print_score d.wcf d.nb_hidden_cells d.nb_marked_cells;;
val mark_cell : demin_cf -> int -> int -> unit = <fun>

# let ending d str =
  draw_field_end d.wcf d.bd ;
  erase_box d.wcf.s_bcf ;
  draw_string_in_box Center str d.wcf.s_bcf Graphics.black
  ignore(Graphics.wait_next_event
    [Graphics.Button_down;Graphics.Key_pressed]);
  raise Fin;;
val ending : demin_cf -> string -> 'a = <fun>
# let reveal d i j =
  let reveal_cell (i,j) =
    d.bd.(i).(j).seen <- true ;
    draw_cell d.wcf d.bd i j ;
    d.nb_hidden_cells <- d.nb_hidden_cells -1
  in

```

```
List.iter reveal_cell (cells_to_see d.bd d.wcf.cf (i,j))
print_score d.wcf d.nb_hidden_cells d.nb_marked_cells ;
if d.nb_hidden_cells = 0 then ending d "GAGNE";;
val reveal : demin_cf -> int -> int -> unit = <fun>
```

La fonction de traitement de l'événement souris filtre une valeur de type *clickon*.

```
# let d_mouse d click = match click with
  Cell (i,j) ->
    if d.bd.(i).(j).seen then ()
    else if d.flag_switch_on then mark_cell d i j
    else if d.bd.(i).(j).flag then ()
    else if d.bd.(i).(j).mined then ending d "PERDU"
    else reveal d i j
  | SelectBox ->
    d.flag_switch_on <- not d.flag_switch_on;
    draw_flag_switch d.wcf d.flag_switch_on
  | Out -> () ;;
val d_mouse : demin_cf -> clickon -> unit = <fun>
```

La création d'une configuration de jeu a besoin de trois paramètres : le nombre de lignes, le nombre de colonnes et le nombre de mines.

```
# let create_demin nb_c nb_r nb_m =
  let nbc = max default_config.nbcols nb_c
  and nbr = max default_config.nbrows nb_r in
  let nbm = min (nbc*nbr) (max 1 nb_m) in
  let cf = { nbcols=nbc ; nbrows=nbr ; nbmines=nbm } in
  generate_seed () ;
  let wcf = make_wcf cf in
  { wcf = wcf ;
    bd = initialize_board wcf.cf;
    nb_marked_cells = 0;
    nb_hidden_cells = cf.nbrows*cf.nbcols-cf.nbmines;
    flag_switch_on = false } ;;
val create_demin : int -> int -> int -> demin_cf = <fun>
```

La fonction de lancement crée une configuration à partir du nombre de colonnes, de lignes et de mines puis démarre la boucle principale de gestion des événements.


```
# let go nbc nbr nbm =  
  let d = create_demin nbc nbr nbm in  
  loop d (d_init d) d_key (d_mouse d) (d_end);;  
val go : int -> int -> int -> unit = <fun>
```

L'appel `go 10 10 10` construit un jeu de la taille de celui de la figure [6.5](#).

Pour en faire plus

On peut faire de ce programme un exécutable autonome. Le chapitre [7](#) explique comment le faire. Une fois cela réalisé, il est pratique de pouvoir donner les tailles du jeu sur la ligne de commande. Le chapitre [8](#) décrit ce passage d'arguments et l'applique au démineur (voir page [??](#)).

Un autre type de problème est de faire jouer la machine pour découvrir les mines. Pour cela il convient de savoir déterminer un coup sûr et de le jouer en priorité, puis de calculer les probabilités de présence d'une mine pour chaque case et de jouer la case de plus faible probabilité.

undefined undefinedundefined undefinedundefined La fonction `fold_left_int` est monomorphe. Le mécanisme de trace connaît le type des arguments et peut alors les afficher lors de leur passage.
undefined

Part II Outils de Développement

Présentation de la partie II

L'ensemble des éléments de l'environnement que nous décrivons est inclus dans la distribution du langage. On y trouve les différents compilateurs, de nombreuses bibliothèques, des outils d'analyse de programmes, des outils pour les analyses lexicale et syntaxique et une interface avec le langage C.

Objective CAML est un langage compilé qui offre deux types de génération de code :

1. du *code-octet* (byte-code) destiné à être exécuté par une *machine virtuelle* ;
2. du *code natif* destiné à être exécuté directement par un micro-processeur.

La boucle d'interaction d'Objective CAML utilise du code-octet pour exécuter les phrases qui lui sont proposées. Elle constitue le premier outil d'aide au développement en offrant la possibilité de typer, compiler et tester rapidement les définitions de fonctions. De plus, elle offre un mécanisme de trace visualisant les valeurs des arguments d'appel et les valeurs de retour des fonctions.

D'autres outils usuels de développement sont également fournis avec la distribution : calcul de dépendances entre fichiers, mise au point (debug) et analyse (profiling). L'outil de mise au point permet l'exécution pas à pas des programmes, l'utilisation de points d'arrêt et l'inspection de valeurs. L'outil d'analyse donne des mesures sur le nombre d'appels ou le temps passé dans telle fonction ou telle partie

de code. Ces deux outils ne sont disponibles que pour les plateformes Unix.

La richesse d'un langage provient de son noyau mais aussi des bibliothèques, ensemble de programmes réutilisables, qui viennent avec lui. Objective CAML n'échappe pas à la règle. Nous avons déjà largement évoqué la bibliothèque graphique venant avec la distribution. Il en est bien d'autres que nous décrirons. Les bibliothèques apportent de nouvelles fonctionnalités au langage, mais elles ne sont pas sans contre-partie. En particulier, elles peuvent présenter une difficulté vis-à-vis de la discipline des types.

Quelque riche que soit l'ensemble des bibliothèques d'un langage, il lui est toujours nécessaire de pouvoir communiquer avec un autre langage. La distribution d'Objective CAML contient une interface avec le langage C qui permet d'appeler des fonctions C à partir d'Objective CAML ou d'être appelé par celui-ci. La difficulté de compréhension et de mise en oeuvre de cette interface tient à ce que les modèles mémoire d'Objective CAML et C sont différents. La raison essentielle de cette différence est qu'un programme Objective CAML intègre un mécanisme de récupération automatique de mémoire.

Aussi bien C qu'Objective CAML permettent l'allocation dynamique de mémoire donc une gestion fine de l'espace selon les besoins d'un programme. Cela n'a de sens que si l'on sait récupérer l'espace inutilisé pour un autre usage au cours d'une exécution. La récupération automatique décharge le programmeur de la gestion de la récupération, source fréquente d'erreurs à l'exécution. Ce trait constitue un des éléments de la sûreté du langage Objective CAML.

Cependant, ce mécanisme a une incidence sur la représentation des données. Aussi, pour utiliser correctement la communication entre le monde Objective CAML et le monde C, il est indispensable de connaître les principes guidant la gestion mémoire.

Le chapitre 7 présente les éléments de base du système Objective CAML : machine virtuelle, compilateurs et bibliothèque d'exécution. Il décrit les différents modes de compilation du langage et compare la portabilité à l'efficacité.

Le chapitre 8 donne une vision globale de l'ensemble des types, fonctions et exceptions prédéfinis venant avec la distribution du système. Il ne remplace pas la lecture du manuel de référence ([LRVD99]) qui décrit très bien ces bibliothèques. Par contre il insiste sur les nouvelles fonctionnalités apportées par certaines d'entre elles. En particulier on peut citer le formatage de sortie, la persistance des valeurs et l'interface avec le système d'exploitation.

Le chapitre 9 présente différentes méthodes de récupération automatique de mémoire pour ensuite décrire le mécanisme utilisé par Objective CAML.

Le chapitre 10 présente les outils de mise au point des programmes Objective CAML. Bien qu'encore assez frustes par certains côtés, ces outils permettent bien souvent de comprendre les dysfonctionnements d'un programme.

Le chapitre 11 décrit les différentes approches des problèmes d'analyse lexicale et syntaxique du langage : bibliothèque de gestion des expressions régulières, outils ocamllex et ocaml yacc, mais aussi utilisation des flots (streams).

Le chapitre 12 décrit l'interface avec le langage C. Il n'est plus possible pour un langage d'être complètement isolé des autres langages. Cette interface autorise un programme Objective CAML à appeler une fonction C, en lui passant des valeurs du monde Objective CAML, et réciproquement. La principale difficulté de cette interface provient du modèle mémoire. Pour cela il est recommandé de lire le chapitre 9 auparavant.

Le chapitre 13 propose deux applications : une bibliothèque graphique enrichie reposant sur un modèle hiérarchique de composants graphiques inspiré de l'AWT² JAVA ; un programme classique de recherche de chemin de moindre coût dans un graphe utilisant notre nouvelle interface graphique ainsi qu'un mécanisme de mémoire-cache.

Modes de compilation et portabilité

Introduction

La transformation du texte d'un programme vers une forme exécutable nécessite plusieurs étapes de transformation. Le regroupement de celles-ci constitue le processus dit de compilation. Celui-ci produit un arbre de syntaxe abstraite (comme pour l'interprète BASIC, page ??) puis une suite d'instructions pour une machine réelle ou virtuelle. Dans ce dernier cas, il est nécessaire d'utiliser un interprète des instructions de cette machine virtuelle. Dans tous les cas, ce que produit le compilateur doit être relié avec la bibliothèque d'exécution qui est fournie avec la distribution du compilateur et qui peut varier selon les processeurs et les systèmes d'exploitation. Cette bibliothèque d'exécution comprend les fonctions primitives (comme les opérations sur les nombres, l'interface avec le système d'exploitation) et le gestionnaire de mémoire. Le langage Objective CAML possède deux compilateurs. Le premier est un compilateur vers une machine virtuelle, il produit du code-octet (byte-code). Le deuxième compilateur est un compilateur vers du code natif qui produit des instructions de processeurs réels, comme les machines Intel, Motorola, SPARC, HP-PA, Power-PC et Alpha. Le compilateur de code-octet privilégie la portabilité alors que le compilateur natif augmente la rapidité d'exécution. La boucle d'interaction, vue dans la première partie de cet ouvrage, utilise le compilateur de code-octet : chaque phrase écrite est compilée puis exécutée dans l'environnement des symboles déjà définis lors d'une session d'interaction.

Plan du chapitre

Ce chapitre présente les différents modes de compilation d'un programme Objective CAML et compare leurs caractéristiques de portabilité et d'efficacité. La première section explique les différentes étapes des compilateurs Objective CAML. La deuxième section décrit les différents modes de compilation et leur syntaxe pour la production d'exécutables. La troisième section montre la construction d'exécutables autonomes, c'est-à-dire indépendants de

l'installation du système Objective CAML. Enfin la quatrième section compare ces différents modes de compilation par rapport aux caractéristiques de portabilité et d'efficacité d'exécution.

Étapes de compilation

L'obtention d'un exécutable passe par les étapes de traduction et d'assemblage décrites par la figure 7.1.

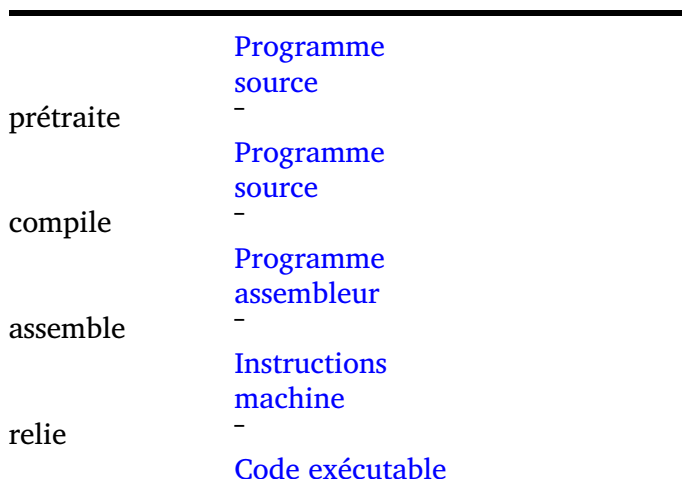


Figure 7.1 : Schéma pour la production d'un exécutable

Le prétraitement remplace des parties de texte par d'autres textes grâce à un système de macros. La compilation traduit le programme source en instructions assembleur. L'assemblage effectue la traduction vers des instructions machines. Enfin l'édition de liens permet d'ajouter la bibliothèque d'exécution, contenant principalement la gestion mémoire, et de faire la jonction avec le système d'exploitation pour les primitives.

Les compilateurs d'Objective CAML

Nous donnons figure 7.2 le détail des phases de génération de code des compilateurs d'Objective CAML. On appelle langage intermédiaire (LI) la représentation interne au compilateur de la suite d'instructions à engendrer.

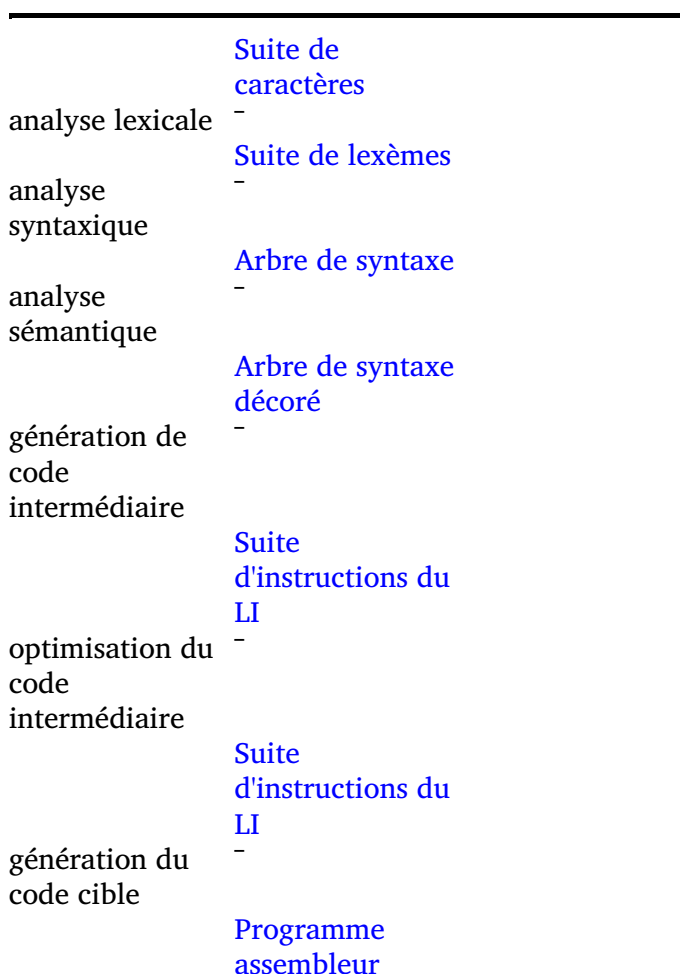


Figure 7.2 : Étapes d'un compilateur

L'analyse lexicale transforme la suite de caractères en une suite d'unités lexicales (lexèmes). Les lexèmes correspondent principalement aux entiers, nombres flottants, caractères, chaînes de caractères et identificateurs. Le message `Illegal character` provient de cette analyse.

L'analyse syntaxique construit un arbre de syntaxe en vérifiant que la suite de lexèmes est correcte par rapport à la grammaire du langage. Les messages `Syntax error` indiquent que la phrase analysée ne respecte pas la grammaire du langage.

L'analyse sémantique, qui parcourt l'arbre de syntaxe, s'intéresse à un autre aspect de la correction du programme. En Objective CAML cette analyse consiste principalement en l'inférence de types qui, si elle réussit, produit le type le plus général d'une expression ou d'une déclaration. Les messages d'erreurs de typage proviennent de cette passe. De même les indications qu'un membre d'une séquence n'est pas du type *unit* sont déterminées à ce moment là. D'autres indications sont données par cette phase, comme l'analyse des filtrages (filtrages non exhaustifs, partie de filtrage non utilisée).

La génération et l'optimisation du code intermédiaire n'engendrent pas de message d'erreur ou d'avertissement. Ces phases de manipulation d'une structure intermédiaire permettent de factoriser le développement des différents compilateurs d'Objective CAML.

La génération du programme cible est l'étape finale du processus de compilation. Elle diffère selon les compilateurs.

Description du compilateur de code-octet

La machine virtuelle d'Objective CAML s'appelle la *Zinc* (pour « *Zinc Is Not Caml* »). Elle est due à Xavier Leroy et est décrite dans ([[Ler90](#)]). Son nom a été choisi pour marquer sa différence d'avec les premières implantations du langage Caml reposant sur la machine virtuelle CAM (pour Categorical Abstract Machine, voir [[CCM87](#)]).

Nous donnons figure [7.3](#) la description des trois éléments permettant l'utilisation du compilateur de code-octet. La première

partie de cette figure comprend l'interprète de la machine Zinc lié à la bibliothèque d'exécution. La deuxième partie correspond au compilateur Objective CAML qui produit du code-octet pour la machine Zinc. La troisième partie contient l'ensemble des bibliothèques venant avec le compilateur. Elles seront décrites au chapitre 8.

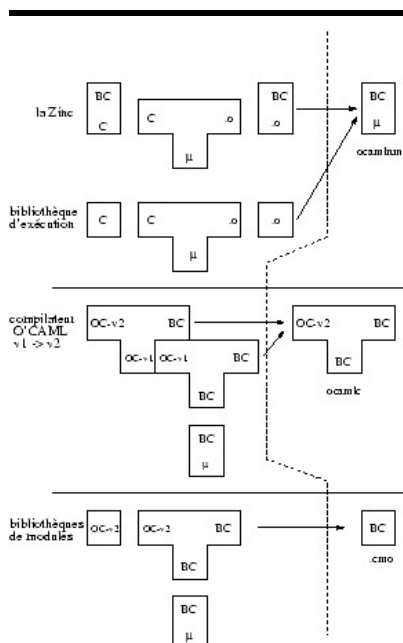


Figure 7.3 : Machine virtuelle

La notation graphique utilisée pour décrire les composants de la figure 7.3 et leur agencement est standard dans le monde de la compilation. Une boîte simple représente un fichier écrit dans le langage indiqué dans la boîte. Une boîte double représente l'interprétation d'un langage par un programme écrit dans un autre. Une boîte triple indique qu'un langage source est compilé vers un langage cible en utilisant un compilateur écrit dans un troisième langage. La figure 7.4 donne la légende de chacune des boîtes.

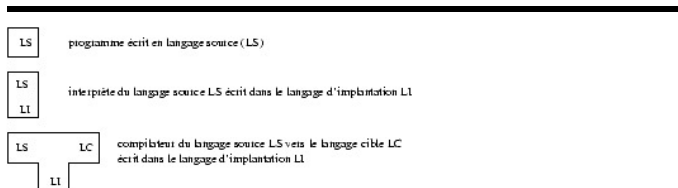


Figure 7.4 : Notation graphique pour les interprètes et les compilateurs

La légende de la figure 7.3 est la suivante :

- BC : code-octet de la Zinc ;
- C : code C ;
- .o : code objet, dépendant de l'architecture ;
- μ : micro-processeur ;
- OC (v1 ou v2) : code Objective CAML.

Remarque

Le compilateur Objective CAML est écrit pour sa plus grande partie en Objective CAML. La deuxième partie de la figure 7.3 montre comment passer d'une version v1 du compilateur vers une version v2.

Modes de compilation

La distribution d'un langage dépend du processeur et du système d'exploitation. Pour chaque architecture (couple processeur, système d'exploitation), une distribution d'Objective CAML contient la boucle d'interaction, le compilateur de code-octet et dans la plupart des cas le compilateur natif sur cette architecture.

Noms des commandes

La figure 7.5 donne les noms des commandes des différents

compilateurs que l'on rencontre dans les distributions d'Objective CAML. Les quatre premières commandes sont disponibles pour toute distribution.

<code>ocaml</code>	boucle d'interaction
<code>ocamlrun</code>	interprète de code-octet
<code>ocamlc</code>	compilateur en ligne de code-octet
<code>ocamlopt</code>	compilateur en ligne de code natif
<code>ocamlc.opt</code>	compilateur en ligne optimisé de code-octet
<code>ocamlopt.opt</code>	compilateur en ligne optimisé de code natif
<code>ocamlmktop</code>	constructeur de nouvelles boucles d'interaction

Figure 7.5 : Commandes de compilation

Les compilateurs optimisés ont été eux-mêmes compilés avec le compilateur natif, produisant un exécutable plus rapide.

Unité de compilation

L'unité de compilation correspond à la plus petite décomposition d'un programme Objective CAML pouvant être compilée. Pour la boucle d'interaction, l'unité de compilation correspond à une phrase du langage. Par contre pour les compilateurs en ligne, l'unité de compilation est un couple de fichiers : le fichier source et le fichier interface. Le fichier interface est optionnel. S'il n'existe pas, alors toutes les déclarations globales du fichier source seront visibles par une autre unité de compilation. La construction des fichiers d'interface sera décrite dans le chapitre sur la programmation modulaire (14). La liaison entre ces deux fichiers utilise les conventions de nommage des extensions de fichiers.

Nommage des extensions de fichiers

La figure 7.6 présente les extensions des différents fichiers utilisés pour les programmes Objective CAML et C.

extension	signification
.ml	fichier source
.mli	fichier interface
.cmo	fichier objet (code-octet)
.cma	fichier d'une bibliothèque objet (code-octet)
.cmi	fichier interface compilé
.cmx	fichier objet (natif)
.cmxa	fichier d'une bibliothèque objet (natif)
.c	fichier source C
.o	fichier objet C (natif)
.a	fichier d'une bibliothèque objet C (natif)

Figure 7.6 : Extension des fichiers

Les fichiers `exemple.ml` et `exemple.mli` forment une unité de compilation. Le fichier d'interface compilé (`exemple.cmi`) est commun au compilateur de code-octet et au compilateur natif. Les fichiers provenant du monde C sont utilisés pour l'interfaçage de Objective CAML avec des bibliothèques C ([12](#)).

Compilateur de code-octet

La forme générale de la commande d'un compilateur en ligne est la suivante :

Objective CAML suit la même règle. On tapera, par exemple :

```
ocamlc -c exemple.ml
```

Les options passées au compilateur sur la ligne de commande suivent les conventions du système Unix. Elles sont précédées du caractère `-`. Les extensions des fichiers sont interprétées de la manière décrite à la figure 7.6. Dans l'exemple ci-dessus, le fichier `exemple.ml` est considéré comme un fichier source Objective CAML et sera donc compilé, ce qui produira les fichiers `exemple.cmo` et `exemple.cmi` car l'option `-c` indique au compilateur de n'engendrer que le code objet. Sans cette option, le compilateur `ocamlc` produit aussi un fichier exécutable `a.out`, c'est-à-dire qu'il effectue aussi l'édition de liens.

Le tableau de la figure 7.7 décrit les principales options du compilateur de code-octet. Le tableau de la figure 7.8 indique les autres options possibles.

Principales options

<code>-a</code>	construit une bibliothèque
<code>-c</code>	compile sans faire l'édition de liens
<code>-o nom_executable</code>	spécifie le nom de l'exécutable
<code>-linkall</code>	indique de lier avec toutes les bibliothèques utilisées
<code>-i</code>	affiche toutes les déclarations globales compilées
<code>-pp commande</code>	utilise <i>commande</i> comme pré-processeur
<code>-unsafe</code>	bascule en mode sans vérification des indices
<code>-v</code>	affiche la version du compilateur
<code>-w liste</code>	choisit selon <i>liste</i> le niveau des messages d'avertissement (voir fig.

	7.9)
-impl <i>fichier</i>	indique que <i>fichier</i> est un source Caml (.ml)
-intf <i>fichier</i>	indique que <i>fichier</i> est une interface Caml (.mli)
-I <i>catalogue</i>	ajoute <i>catalogue</i> dans la liste des catalogues

Figure 7.7 : Principales options du compilateur de code-octet

Autres options	
processus légers	-thread (19, page ??)
mise au point	-g, -noassert (10, page ??)
exécutables autonomes	-custom, -cclib, -ccopt, -cc (voir page ??)
<i>runtime</i>	-make-runtime , -use-runtime
interface C	-output-obj (12, page ??)

Figure 7.8 : Autres options du compilateur de code-octet

Pour obtenir l'affichage de l'ensemble des options du compilateur de code-octet, il suffit de lui passer l'option -help.

Les niveaux de message d'avertissement (warning) sont décrits à la figure 7.9. Un niveau de message est une bascule (activée/désactivée) représentée par une lettre. Une majuscule active ce niveau et une minuscule le désactive.

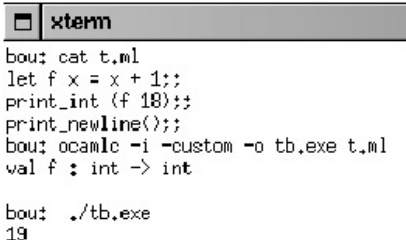
Principaux niveaux	
A/a	active/désactive tous les

F/f	messages application partielle dans une séquence
P/p	pour les filtrages incomplets
U/u	pour les cas inutiles d'un filtrage
X/x	active/désactive tous les autres messages
pour la couche objet	M/m et V/v (voir chapitre 15)

Figure 7.9 : Description des avertissements de compilation

Par défaut le niveau maximal (A) est choisi par le compilateur.

Un exemple d'utilisation du compilateur de code-octet est donné figure 7.10.



```

xterm
bou: cat t.ml
let f x = x + 1;;
print_int (f 18);;
print_newline();;
bou: ocamlc -i -custom -o tb.exe t.ml
val f : int -> int

bou: ./tb.exe
19

```

Figure 7.10 : Session avec le compilateur de code-octet

Compilateur natif

Le compilateur natif possède un comportement proche de celui du compilateur de code-octet bien que les fichiers produits soient différents. Les options de compilation sont globalement les mêmes que celles décrites aux figures 7.7 et 7.8. Il faut cependant retirer les options liées au *runtime* de la figure 7.8. Les options spécifiques au compilateur natif sont données à la figure 7.11. Les différents niveaux de *warning* sont les mêmes.

-compact	optimise en espace le code produit
-S	conserve le code assembleur dans un fichier
-inline <i>niveau</i>	indique le grain de traduction d'une fonction à plat

Figure 7.11 : Options particulières pour le compilateur natif.

L'expansion en ligne (inlining) est une version élaborée de la macro-expansion du prétraitement. Elle remplace l'appel d'une fonction par le corps de la fonction où les arguments sont fixés. Plusieurs appels différents correspondront à une nouvelle traduction du corps de la fonction. On évite ainsi d'exécuter la séquence d'appel de fonction, mais la taille du code peut s'en trouver accrue. Les principaux niveaux d'inlining sont :

- 0 : l'expansion n'est autorisée que lorsqu'elle n'accroît pas la taille du code ;
- 1 : c'est la valeur par défaut, elle accepte une légère augmentation de la taille du code ;
- $n > 1$: augmente la tolérance d'accroissement de la taille du code expansé.

Boucle d'interaction

La boucle d'interaction ne possède que deux options pour son lancement :

- -I *catalogue* : qui ajoute le chemin indiqué en tête de la liste des chemins de recherche des fichiers sources ou compilés ;
- -unsafe : qui indique au compilateur de ne plus tester les débordements d'indice de tableaux ou de chaînes de caractères.

La boucle d'interaction possède plusieurs directives de compilation qui permettent de modifier interactivement son comportement. Elles sont décrites à la figure 7.12. Toutes les directives commencent par le caractère # et se terminent par les habituels ; ; .

#quit ; ;	sort de la boucle d'interaction
#directory <i>catalogue</i> ; ;	ajoute un chemin en tête
#cd <i>catalogue</i> ; ;	change de catalogue courant
#load <i>fichier_objet</i> ; ;	charge un fichier .cmo
#use <i>fichier_source</i> ; ;	compile et charge un fichier source
#print_depth <i>profondeur</i> ; ;	modifie la profondeur d'affichage
#print_length <i>largeur</i> ; ;	idem pour la largeur
#install_printer <i>fonction</i> ; ;	spécifie une fonction d'affichage
#remove_printer <i>fonction</i> ; ;	repassse à l'afficheur standard
#trace <i>fonction</i> ; ;	trace les arguments d'une fonction
#untrace <i>fonction</i> ; ;	enlève la trace d'une fonction
#untrace_all ; ;	enlève toutes les traces

Figure 7.12 : Directives de la boucle d'interaction

Les directives sur les catalogues respectent les conventions du système d'exploitation utilisé.

Les directives de chargement n'ont pas exactement le même comportement. La directive `#use` lit le fichier source indiqué comme s'il avait été tapé directement en interactif. La directive `#load` charge le fichier d'extension `.cmo` indiqué. Dans ce dernier cas, les déclarations globales de ce fichier ne sont pas directement accessibles.

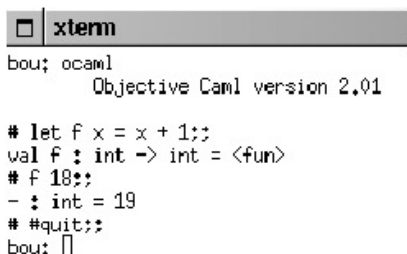
Pour cela il faut utiliser la notation pointée. Si le fichier `exemple.ml` contient la déclaration globale `f`, alors une fois chargé le code-octet (`#load "exemple.cmo";;`), on accédera à la valeur de `f` par `Exemple.f`, où la première lettre du fichier est remplacée par une majuscule. Cette notation vient du système de modules d'Objective CAML (voir chapitre 14, page ??).

Les directives de paramétrage de la profondeur et de la largeur d'affichage permettent de contrôler l'affichage des valeurs. Ce qui est utile quand des valeurs volumineuses doivent être affichées.

Les directives de redéfinition des fonctions d'impression pour des types donnés permettent de définir leur présentation. Pour qu'elles s'intègrent bien avec l'affichage par défaut, il sera nécessaire d'utiliser la bibliothèque `Format` (8) pour les définir.

Les directives de traçage des arguments et du résultat de fonctions sont particulièrement utiles pour la mise au point des programmes. Elles seront manipulées au chapitre sur l'analyse de programme (10).

La figure 7.13 montre une session avec la boucle d'interaction.

A screenshot of an xterm window titled 'xterm'. The terminal shows an OCaml interactive session. The prompt 'bou:' is followed by 'ocaml', which outputs 'Objective Caml version 2.01'. Then, the user enters a series of commands: '# let f x = x + 1;;', 'val f : int -> int = <fun>', '# f 18;;', '- : int = 19', '# #quit;;', and finally 'bou:' followed by a cursor.

```

xterm
bou: ocaml
      Objective Caml version 2.01

# let f x = x + 1;;
val f : int -> int = <fun>
# f 18;;
- : int = 19
# #quit;;
bou: 
```

Figure 7.13 : Session avec la boucle d'interaction

Construction de nouvelles boucles d'interaction

La commande `ocamlmktop` permet de construire de nouvelles boucles d'interaction dans lesquelles un certain nombre de bibliothèques ou modules sont préchargés. Outre le fait de ne plus avoir besoin d'effectuer les `#load` en début de session, cela est indispensable pour intégrer des bibliothèques écrites en C car il est alors nécessaire de modifier la bibliothèque d'exécution.

Les options de cette commande sont un sous-ensemble des options du compilateur de code-octet (`ocamlc`) :

`-cclib libname`, `-ccopt option`, `-custom`, `-I catalogue` et `-o nom_executable`

Le chapitre sur la programmation graphique (5, page ??) utilise cette commande pour construire un toplevel contenant la bibliothèque Graphics de la manière suivante :

```
ocamlmktop -custom -o mytoplevel graphics.cma -cclib \
           -L/usr/X11/lib -cclib -lX11
```

Cette commande construit l'exécutable de nom `mytoplevel`, contenant la bibliothèque de code-octet `graphics.cma`. Cet exécutable autonome (`-custom` (voir section suivante)) sera lié à la

bibliothèque X11 (libX11.a) qui elle-même sera recherchée dans le chemin `/usr/X11/lib`.

Exécutables autonomes

Un exécutable autonome (standalone) est un exécutable indépendant de l'installation d'Objective CAML sur la machine utilisée. D'une part, cela facilite la diffusion d'une application sous forme binaire. D'autre part si l'on installe une nouvelle version d'Objective CAML à un emplacement différent, cet exécutable continuera à fonctionner.

Le compilateur natif produit automatiquement des exécutables autonomes. En revanche, sans l'option `-custom`, le compilateur de code-octet produit un exécutable qui a besoin de l'interprète `ocamlrun`. Soit le fichier `exemple.ml` suivant :

```
let f x = x + 1;;  
print_int (f 18);;  
print_newline();;
```

La commande suivante produit le fichier `exemple.exe` :

```
ocamlc -o exemple.exe t.ml
```

Ce fichier d'une taille de 8ko peut être exécuté en le passant comme paramètre à l'interprète de code-octet :

```
$ ocamlrun exemple.exe  
19
```

L'interprète exécute les instructions de la machine Zinc contenues dans le fichier `exemple.exe` et utilise le chemin d'accès standard aux bibliothèques.

Sous Unix, la première ligne du fichier `exemple.exe` contient la localisation de l'interprète de code-octet, par exemple :

```
#!/usr/local/bin/ocamlrun
```

Cette indication permet de lancer directement l'exécution du fichier

exemple.exe par l'interprète de commandes. Son exécution lance la commande décrite par cette première ligne. Il faut donc bien trouver la commande `ocamlrun` à l'emplacement indiqué, sinon le message d'erreur `Command not found` apparaîtra et l'exécution sera arrêtée.

La même compilation avec l'option `-custom` produit un exécutable autonome de nom `exauto.exe` :

```
ocamlc -custom -o exaauto.exe t.ml
```

Sa taille est d'environ 85Ko car il contient, outre le code-octet du programme, l'interprète de ce code-octet. Ce fichier peut être exécuté directement ou copié sur une autre machine (de même architecture et de même système) pour être exécuté.

Warning

La taille du fichier d'un exécutable ne correspond pas à la taille du processus de son exécution. En effet le fichier ne tient pas compte de l'allocation dynamique de mémoire du programme (9).

Portabilité et efficacité

L'intérêt de compiler vers une machine abstraite est de produire un exécutable indépendant de l'architecture de la machine réelle où il tourne. Son principal inconvénient provient de l'interprétation de ces instructions virtuelles. Un compilateur natif produira un code plus efficace, mais ne pouvant s'exécuter que sur un seul type d'architecture. Il est donc souhaitable d'avoir le choix entre ces deux types de compilation selon le type d'application développée. L'autonomie d'un exécutable, c'est-à-dire l'indépendance de celui-ci de l'installation de la distribution d'Objective CAML, fait aussi perdre la propriété de portabilité.

Autonomie et portabilité

Pour produire un exécutable autonome, le compilateur de code-

octet a effectué une édition de liens du code-octet t.cmo avec la bibliothèque d'exécution, l'interprète de code-octet et une amorce de programme écrite en C. Cela sous-entend qu'il y avait un compilateur C sur la machine hôte. À ce moment là même si la partie importante de l'exécutable contient les instructions en code-octet, cet exécutable n'est plus portable sur d'autres systèmes ou d'autres architectures machine.

Cela n'était pas le cas pour la version non autonome. En effet les instructions de code-octet ne diffèrent pas d'une machine à une autre, car la machine Zinc n'existant pas, seul son interprète compte. Si l'interprète de code-octet existe sur des machines d'architecture ou de système différents, rien ne les empêche d'exécuter les instructions de code-octet produites par d'autres compilateurs Objective CAML en utilisant la commande `ocamlrun`. Celle-ci fait partie des différentes distributions d'Objective CAML pour Sparc sous Solaris, Intel sous Windows, etc. Cependant, il est toujours préférable d'utiliser des versions de l'interprète de code-octet et du compilateur de même numéro.

La portabilité des fichiers objets en code-octet autorise la diffusion de bibliothèques directement sous cette forme et immédiatement utilisables sur les machines possédant l'interprète de code-octet.

Efficacité d'exécution

Le compilateur de code-octet produit une suite d'instructions de la machine Zinc, qui, au moment de l'exécution, seront interprétées par `ocamlrun`. Cette phase d'interprétation a un certain coût pour la rapidité d'exécution du programme. On peut se représenter l'interprète de code-octet comme une grande structure d'aiguillage (filtrage match ... with) où chaque instruction est un motif et les branches de calcul modifient la pile et le compteur ordinal (adresse de la prochaine instruction). Ce filtrage, bien qu'optimisé, induit une perte de performance.

Bien que ne testant pas toutes les parties du langage, le petit exemple suivant qui calcule la fonction de Fibonacci montre les différences de temps d'exécution entre le compilateur de code-octet et le compilateur natif. Soit le programme `fib.ml` suivant :

```
let rec fib n =  
  if n < 2 then 1  
  else (fib (n-1)) + (fib(n-2));;
```

et le programme main.ml suivant :

```
for i = 1 to 10 do  
  print_int (Fib.fib 30 );  
  print_newline()  
done;;
```

et leurs compilations suivantes :

```
$ ocamlc -o fib.exe fib.ml main.ml  
$ ocamlpt -o fibopt.exe fib.ml main.ml
```

produisant les exécutables fib.exe et fibopt.exe. On obtient avec la commande Unix time sur un Pentium 350 sous Linux les temps suivants :

fib.exe (code-octet)	fibopt.exe (natif)
7 s	1 s

Ce qui correspond à un facteur 7 entre les deux versions du même programme. Ce programme ne teste pas toutes les caractéristiques du langage. Le gain dépend fortement du type d'application.

Exercices

Création de toplevels et d'exécutables autonomes

On reprend l'interprète de Basic pour créer une nouvelle boucle d'interaction.

1. Découper l'application Basic en 4 fichiers d'extension `.ml` contenant respectivement : la syntaxe abstraite ([syntax.ml](#)), l'afficheur ([pprint.ml](#)), l'analyse syntaxique ([alexsynt.ml](#)) et l'évaluation des instructions ([eval.ml](#)). Chaque début de

fichier contiendra l'ouverture des modules nécessaires à sa compilation.

```
type op_unr = OPPOSE | NON ;;
```

```
type op_bin = PLUS | MOINS | MULT | DIV | MOD  
| EGAL | INF | INFEQ | SUP | SUPEQ | DIFF  
| ET | OU ;;
```

```
type expression =  
  ExpInt of int  
  | ExpVar of string  
  | ExpStr of string  
  | ExpUnr of op_unr * expression  
  | ExpBin of expression * op_bin * expression ;;
```

```
type instruction =  
  Rem of string  
  | Goto of int  
  | Print of expression  
  | Input of string  
  | If of expression * int  
  | Let of string * expression ;;
```

```
type ligne = { num : int ; inst : instruction } ;;
```

```
type program = ligne list ;;
```

```
type phrase = Ligne of ligne | List | Run | End ;
```

```
let priority_ou = function NON -> 1 | OPPOSE -> 7  
let priority_ob = function  
  MULT | DIV -> 6  
  | PLUS | MOINS -> 5  
  | MOD -> 4  
  | EGAL | INF | INFEQ | SUP | SUPEQ | DIFF -> 3  
  | ET | OU -> 2 ;;
```

```
let pp_opbin = function
```



```

PLUS -> "+" | MULT -> "*" | MOD -> "%" | MOINS -> "-"
| DIV -> "/" | EGAL -> "=" | INF -> "<"
| INFEQ -> "<=" | SUP -> ">"
| SUPEQ -> ">=" | DIFF -> "<>" | ET -> "&"
let pp_opunr = function OPPOSE -> "-" | NON -> "!"

```

```

open Syntax;;

```

```

let parenthese x = "(" ^ x ^ ")";;

```

```

let pp_expression =
  let rec ppg pr = function
    ExpInt n -> (string_of_int n)
    | ExpVar v -> v
    | ExpStr s -> "\"" ^ s ^ "\""
    | ExpUnr (op,e) ->
      let res = (pp_opunr op)^(ppg (priority_of op) e)
      in if pr=0 then res else parenthese res
    | ExpBin (e1,op,e2) ->
      let pr2 = priority_of op
      in let res = (ppg pr2 e1)^(pp_opbin op)^(ppd pr2 e2)
      (* parenthèse si la priorité n'est pas supérieure *)
      in if pr2 >= pr then res else parenthese res
      and ppd pr exp = match exp with
        (* les sous-arbres droits ne diffèrent *)
        (* que pour les opérateurs binaires *)
        ExpBin (e1,op,e2) ->
          let pr2 = priority_of op
          in let res = (ppg pr2 e1)^(pp_opbin op)^(ppd pr2 e2)
          in if pr2 > pr then res else parenthese res
          | _ -> ppg pr exp
      in ppg 0 ;;

```

```

let pp_instruction = function
  Rem s -> "REM " ^ s
  | Goto n -> "GOTO " ^ (string_of_int n)
  | Print e -> "PRINT " ^ (pp_expression e)
  | Input v -> "INPUT " ^ v
  | If (e,n) -> "IF "^(pp_expression e)^" THEN "^(

```

```

| Let (v,e) -> "LET " ^ v ^ " = " ^ (pp_expressio

let pp_ligne l = (string_of_int l.num) ^ " " ^ (pp

open Syntax;;

type lexeme = Lint of int
| Lident of string
| Lsymbol of string
| Lstring of string
| Lfin ;;

type chaine_lexer = {chaine:string; mutable courant

let init_lex s = { chaine=s; courant=0 ; taille=Str

let avance cl = cl.courant <- cl.courant+1 ;;

let avance_n cl n = cl.courant <- cl.courant+n ;;

let extrait pred cl =
  let st = cl.chaine and ct = cl.courant in
  let rec ext n = if n<cl.taille && (pred st.[n]) th
  let res = ext ct
  in cl.courant <- res ; String.sub cl.chaine ct (re

let extrait_int =
  let est_entier = function '0'..'9' -> true | _ ->
  in function cl -> int_of_string (extrait est_entie
let extrait_ident =
  let est_alpha_num = function
  'a'..'z' | 'A'..'Z' | '0' .. '9' | '_' -> true
  | _ -> false
  in extrait est_alpha_num ;;

exception LexerErreur ;;

let rec lexer cl =
  let lexer_char c = match c with

```

```

| '\t'      -> avance cl ; lexer cl
| 'a'..'z'
| 'A'..'Z' -> Lident (extrait_ident cl)
| '0'..'9' -> Lint (extrait_int cl)
| '"'      -> avance cl ;
let res = Lstring (extrait ((<>) '"') cl)
in avance cl ; res
| '+' | '-' | '*' | '/' | '%' | '&' | '|' | '!' |
avance cl; Lsymbol (String.make 1 c)
| '<'
| '>'      -> avance cl;
if cl.courant >= cl.taille then Lsymbol (String.ma
else let cs = cl.chaine.[cl.courant]
in ( match (c,cs) with
('<','=') -> avance cl; Lsymbol "<="
| ('>','=') -> avance cl; Lsymbol ">="
| ('<','>') -> avance cl; Lsymbol "<>"
| _         -> Lsymbol (String.make 1 c) )
| _ -> raise LexerErreur
in
if cl.courant >= cl.taille then Lfin
else lexer_char cl.chaine.[cl.courant] ;;

type exp_elem =
  Texp of expression      (* expression          *)
  | Tbin of op_bin        (* opérateur binaire  *)
  | Tunr of op_unr       (* opérateur unaire   *)
  | Tpg                  (* parenthèse gauche *) ;;

exception ParseErreur ;;

let symb_unr = function
  "!" -> NON | "-" -> OPPOSE | _ -> raise ParseErreur
let symb_bin = function
  "+" -> PLUS | "-" -> MOINS | "*" -> MULT | "/" ->
  | "=" -> EGAL | "<" -> INF | "<=" -> INFEQ | ">" -
  | ">=" -> SUPEQ | "<>" -> DIFF | "&" -> ET | "|" -
  | _ -> raise ParseErreur
let tsymb s = try Tbin (symb_bin s) with ParseErreur

```

```

let reduit pr = function
  (Texp e)::(Tunr op)::st when (priority_ou op) >= p
-> (Texp (ExpUnr (op,e)))::st
| (Texp e1)::(Tbin op)::(Texp e2)::st when (priori
-> (Texp (ExpBin (e2,op,e1)))::st
| _ -> raise ParseErreur ;;

let rec empile_ou_reduit lex stack = match lex , st
Lint n , _ -> (Texp (ExpInt n))::stack
| Lident v , _ -> (Texp (ExpVar v))::stack
| Lstring s , _ -> (Texp (ExpStr s))::stack
| Lsymbol "(" , _ -> Tpg::stack
| Lsymbol ")" , (Texp e)::Tpg::st -> (Texp e)::s
| Lsymbol ")" , _ -> empile_ou_reduit lex (reduit
| Lsymbol s , _
-> let symbole =
if s<>"-" then tsymb s
(* lever l'ambiguïté du symbole ``-' ' *)
(* suivant la pile (i.e dernier exp_elem empile) *
else match stack
with (Texp _)::_ -> Tbin MOINS
| _ -> Tunr OPPOSE
in ( match symbole with
Tunr op -> (Tunr op)::stack
| Tbin op ->
( try empile_ou_reduit lex (reduit (priority_ob op
stack )
with ParseErreur -> (Tbin op)::stack )
| _ -> raise ParseErreur )
| _ , _ -> raise ParseErreur ;;

let rec reduit_tout = function
| [] -> raise ParseErreur
| [Texp x] -> x
| st -> reduit_tout (reduit 0 st) ;;

let parse_exp fin cl =
let p = ref 0
in let rec parse_un stack =

```

```

let l = ( p:=cl.courant ; lexer cl)
in if not (fin l) then parse_un (empile_ou_reduit
else ( cl.courant <- !p ; reduit_tout stack )
in parse_un [] ;;

let parse_inst cl = match lexer cl with
  Lident s -> ( match s with
    "REM" -> Rem (extrait (fun _ -> true) cl)
    | "GOTO" -> Goto (match lexer cl with
      Lint p -> p
      | _ -> raise ParseErreur)
    | "INPUT" -> Input (match lexer cl with
      Lident v -> v
      | _ -> raise ParseErreur)
    | "PRINT" -> Print (parse_exp ((=) Lfin) cl)
    | "LET" ->
      let l2 = lexer cl and l3 = lexer cl
      in ( match l2 ,l3 with
        (Lident v,Lsymbol "=") -> Let (v,parse_exp ((=) Lf
        | _ -> raise ParseErreur )
        | "IF" ->
          let test = parse_exp ((=) (Lident "THEN")) cl
          in ( match ignore (lexer cl) ; lexer cl with
            Lint n -> If (test,n)
            | _ -> raise ParseErreur )
            | _ -> raise ParseErreur )
            | _ -> raise ParseErreur ;;

let parse str =
  let cl = init_lex str
  in match lexer cl with
    Lint n -> Ligne { num=n ; inst=parse_inst cl }
    | Lident "LIST" -> List
    | Lident "RUN" -> Run
    | Lident "END" -> End
    | _ -> raise ParseErreur ;;

open Syntax;;
open Pprint;;

```

```

open Alexsynt;;

type valeur = Vint of int | Vstr of string | Vbool

type environnement = (string * valeur) list ;;

type etat = { ligne:int ; prog:program ; env:enviro

exception RunErreur of int
let runerr n = raise (RunErreur n) ;;

let rec eval_exp n envt expr = match expr with
  ExpInt p  -> Vint p
  | ExpVar v  -> ( try List.assoc v envt with Not_found -> runerr n )
  | ExpUnr (OPPOSE,e) ->
    ( match eval_exp n envt e with
      Vint p -> Vint (-p)
      | _ -> runerr n )
  | ExpUnr (NON,e) ->
    ( match eval_exp n envt e with
      Vbool p -> Vbool (not p)
      | _ -> runerr n )
  | ExpStr s -> Vstr s
  | ExpBin (e1,op,e2)
    -> match eval_exp n envt e1 , op , eval_exp n envt e2 with
      Vint v1 , PLUS , Vint v2 -> Vint (v1 + v2)
      | Vint v1 , MOINS , Vint v2 -> Vint (v1 - v2)
      | Vint v1 , MULT , Vint v2 -> Vint (v1 * v2)
      | Vint v1 , DIV , Vint v2 when v2<>0 -> Vint (v1 / v2)
      | Vint v1 , MOD , Vint v2 when v2<>0 -> Vint (v1 mod v2)
      | Vint v1 , EGAL , Vint v2 -> Vbool (v1 = v2)
      | Vint v1 , DIFF , Vint v2 -> Vbool (v1 <> v2)
      | Vint v1 , INF , Vint v2 -> Vbool (v1 < v2)
      | Vint v1 , SUP , Vint v2 -> Vbool (v1 > v2)
      | Vint v1 , INFEQ , Vint v2 -> Vbool (v1 <= v2)
      | Vint v1 , SUPEQ , Vint v2 -> Vbool (v1 >= v2)
      | Vbool v1 , ET , Vbool v2 -> Vbool (v1 && v2)
      | Vbool v1 , OU , Vbool v2 -> Vbool (v1 || v2)

```

```

| Vstr v1 , PLUS , Vstr v2 -> Vstr (v1 ^ v2)
| _ , _ , _ -> runerr n ;;

```

```

let rec ajoute v e env = match env with
[] -> [v,e]
| (w,f)::l -> if w=v then (v,e)::l else (w,f)::(aj

```

```

let rec goto_ligne n prog = match prog with
[] -> runerr n
| l::ll -> if l.num = n then prog
else if l.num<n then goto_ligne n ll
else runerr n ;;

```

```

let print_valeur v = match v with
Vint n -> print_int n
| Vbool true -> print_string "true"
| Vbool false -> print_string "false"
| Vstr s -> print_string s ;;

```

```

let eval_inst etat =
let lc, ns =
match goto_ligne etat.ligne etat.prog with
[] -> failwith "programme vide"
| lc::[] -> lc,-1
| lc::ls::_ -> lc,ls.num
in
match lc.inst with
Rem _ -> { etat with ligne=ns }
| Print e -> print_valeur (eval_exp lc.num etat.
print_newline () ;
{ etat with ligne=ns }
| Let(v,e) -> let ev = eval_exp lc.num etat.env e
in { etat with ligne=ns; env=ajoute v ev etat.env
| Goto n -> { etat with ligne=n }
| Input v -> let x = try read_int ()
with Failure "int_of_string" -> 0
in { etat with ligne=ns ; env=ajoute v (Vint x) et
| If (t,n) -> match eval_exp lc.num etat.env t with
Vbool true -> { etat with ligne=n }

```

```

| Vbool false -> { etat with ligne=ns }
| _ -> runerr n ;;

let rec run etat =
  if etat.ligne = -1 then etat else run (eval_inst e)

let rec inserer ligne p = match p with
  [] -> [ligne]
  | l::prog ->
    if l.num < ligne.num then l::(inserer ligne prog)
    else if l.num=ligne.num then ligne::prog
    else ligne::l::prog ;;

let print_prog etat =
  let print_ligne x = print_string (pp_ligne x) ; pr
  in print_newline () ;
  List.iter print_ligne etat.prog ;
  print_newline () ;;

let premiere_ligne = function [] -> 0 | i::_ -> i.n

exception Fin
let une_commande etat =
  print_string "> " ; flush stdout ;
  try
    match parse (input_line stdin) with
    Ligne l -> { etat with prog=inserer l etat.prog }
    | List -> (print_prog etat ; etat )
    | Run -> run {etat with ligne = premiere_ligne eta
    | End -> raise Fin
  with
    LexerErreur -> print_string "Illegal character\n";
    | ParseErreur -> print_string "syntax error\n"; et
    | RunErreur n ->
      print_string "runtime error at line ";
      print_int n ;
      print_string "\n";
      etat ;;

let go () =

```



```

try
  print_string "Mini-BASIC version 0.1\n\n";
  let rec loop etat = loop (une_commande etat) in
  loop { ligne=0; prog=[]; env=[] }
with Fin -> print_string "A bientôt...\n";;

```

2. Compiler séparément chacun des fichiers.

```

$ ocamlc -c syntax.ml
$ ocamlc -c pprint.ml
$ ocamlc -c alexsynt.ml
$ ocamlc -c eval.ml

```

3. Rajouter un fichier [mainbasic.ml](#) qui ne contiendra que le lancement de la fonction principale.

```

open Eval;;

```

```

go ();;

```

4. Créer un nouveau toplevel, de nom [topbasic](#), qui démarre l'interprète Basic.

```

$ ocamlmktop -o topbasic syntax.cmo pprint.cmo alex

```

5. Créer un exécutable [autonome](#) lançant l'interprète Basic.

```

$ ocamlc -custom -o basic.exe syntax.cmo pprint.cmo

```

Comparaison de performances

On cherche à comparer les performances du code produit par le

compilateur de byte-code et par le compilateur natif. Pour cela on écrira une application de tri sur des listes et des tableaux.

1. Écrire une fonction polymorphe de tri sur les listes, la relation d'ordre sera passée comme paramètre de la fonction de tri. L'algorithme de tri est laissé au choix du lecteur. On peut imaginer soit un tri à bulle, soit un tri rapide. On écrira cette fonction dans le fichier [sort.ml](#).

```
(*****  
(*  
(*                               Objective Caml  
(*  
(*           Xavier Leroy, projet Cristal, INRIA Rocquencourt  
(*  
(* Copyright 1996 Institut National de Recherche en Informatique et  
(* Automatique.  Distributed only by permission.  
(*  
(*****  
  
(* $Id: sort.mli,v 1.1 2000/01/21 09:40:00 emmanuel  
  
(* Module [Sort]: sorting and merging lists *)  
  
val list : ('a -> 'a -> bool) -> 'a list -> 'a list  
  (* Sort a list in increasing order according to an ordering predicate.  
  The predicate should return [true] if its first argument is  
  less than or equal to its second argument. *)  
  
val array : ('a -> 'a -> bool) -> 'a array -> unit  
  (* Sort an array in increasing order according to an ordering  
  predicate.  
  The predicate should return [true] if its first argument is  
  less than or equal to its second argument.  
  The array is sorted in place. *)  
  
val merge : ('a -> 'a -> bool) -> 'a list -> 'a list  
  (* Merge two lists according to the given ordering predicate.  
  Assuming the two argument lists are sorted according to the  
  predicate. *)
```

predicate, [merge] returns a sorted list containing the elements from the two lists. The behavior is undefined if the argument lists were not sorted. *)

2. Écrire un programme principal, dans le fichier [trilist.ml](#), qui utilise la fonction précédente et l'applique sur une liste d'entiers en la triant en ordre croissant puis en ordre décroissant.

```
let interval order next a b =  
  let rec aux a =  
    if not (order a b) then [a] else a :: aux (next a)  
  in aux a;;
```

3. Créer deux [exécutables](#) autonomes, un avec le compilateur de byte-code et l'autre en natif. Mesurer les temps d'exécution de ces deux programmes. On choisira des listes de tailles suffisamment importantes pour les mesures de temps.

```
ocamlc -custom -o trilbyte.exe sort.mli sort.ml int
```

4. Réécrire le tri pour des tableaux, en utilisant toujours une fonction d'ordre en paramètre. [Effectuer](#) les mesures sur des tableaux remplis de la même manière que les listes précédentes.

```
let main () =  
  let il = Array.of_list(Interval.interval (>)) (fun  
    and il2 = Array.of_list(Interval.interval (<)) (fun  
    Sort.array (<) il, Sort.array (>) il2;;  
  
  main();;
```

5. Que peut-on dire des [résultats](#) des mesures?

Résumé

Ce chapitre a montré les différentes passes des compilateurs Objective CAML. Le compilateur de byte-code favorise la portabilité du code binaire et permet donc de distribuer des bibliothèques sous ce format-là. Cette propriété est perdue dans le cas d'exécutables autonomes. Le compilateur natif privilégie l'efficacité du code produit en perdant aussi la portabilité. Dans ce cas, la production d'exécutables autonomes ne change pas la perte de portabilité.

Pour en savoir plus

Les techniques de compilation vers des machines abstraites ont été utilisées par les premières implantations de SmallTalk, puis ont été reprises pour les langages fonctionnels LISP ou ML. L'argument, réel, de perte d'efficacité due aux machines abstraites a rejeté un temps dans l'ombre cette technique. Cependant, aujourd'hui, la langage JAVA en fait un argument de vente. Les machines abstraites possèdent plusieurs avantages. Le premier est de faciliter le portage d'un compilateur vers différentes architectures. En effet la partie du compilateur à réécrire pour un portage est bien cernée (l'interprète de la machine abstraite et une partie de la bibliothèque d'exécution). D'autre part elles permettent la portabilité du code produit. Il est possible de compiler une application sur une architecture système-machine et de l'exécuter sur une autre. Enfin elles permettent de simplifier l'écriture du compilateur en ajoutant des instructions spécifiques pour le type de langage à compiler. Dans le cas des langages fonctionnels, les machines abstraites permettent de créer facilement des fermetures (couple environnement, code) en ajoutant à la machine abstraite la notion d'environnement d'exécution.

Pour pallier la perte d'efficacité due à l'utilisation d'un interprète de code-octet, on peut procéder à une expansion des instructions de la machine abstraite vers les instructions de la machine réelle au moment de leur chargement. On trouve ce type d'expansion dans les implantations de Le Lisp (llm3) et de JAVA (JIT). Néanmoins ces performances ne peuvent égaler celles d'un compilateur natif ou vers C.

Une difficulté de la compilation des langages fonctionnels provient des fermetures. Ces dernières contiennent en effet le code à exécuter ainsi que l'environnement d'exécution de ce code (voir page ??).

Les choix d'implantation des environnements et de l'accès aux valeurs qu'ils contiennent influencent fortement l'efficacité du code produit. Une optimisation importante de la gestion des environnements consiste à obtenir un accès aux valeurs en temps constant : les variables sont vues comme des indices dans un tableau contenant leur valeur. Cela nécessite un prétraitement des expressions fonctionnelles. On en trouve un premier exemple dans la *Functional Abstract Machine* de L. Cardelli. La Zinc utilise également cette technique. Une autre optimisation cruciale est d'éviter de construire des fermetures inutiles. Bien qu'en ML toute fonction puisse être vue comme une fonction à un argument, il est indispensable de ne pas créer de fermetures intermédiaires lors de l'application de plusieurs arguments. Par exemple, quand on applique deux entiers à la fonction *add*, il n'est pas utile de construire la première fermeture correspondant à la fonction *add* appliquée au premier argument. Il faut noter que la création d'une fermeture nécessite l'allocation d'un certain espace mémoire pour l'environnement donc en allouer trop déclenchera plus souvent le récupérateur automatique de mémoire (9). Or, la récupération automatique de mémoire est le deuxième point sensible pour les performances.

Enfin la propriété d'auto-amorçage (bootstrap) permet d'écrire la plus grande partie du compilateur dans le langage qu'il doit compiler. Pour cela, comme la poule et l'oeuf, il est nécessaire de définir la partie minimale du langage permettant ensuite de l'augmenter. En fait cette propriété est difficilement appréciable pour classer les langages et leurs implantations. Disons que cela sert pour montrer la capacité d'un langage à être langage d'implantation pour un compilateur. Un compilateur est un programme d'une bonne taille et l'auto-amorçage est un bon test de la correction du compilateur qui permet d'évaluer ses performances. Donnons comme référence le lien suivant :

Lien: <http://caml.inria.fr/camlstone.txt>

On y trouve les temps de compilation du langage Caml sur une cinquantaine de machines. Ces temps sont pour des versions plus anciennes qu'Objective CAML mais permettent de se faire une idée de l'efficacité d'un couple machine-système pour Objective CAML.

Bibliothèques

Introduction

Tout langage vient avec un ensemble de programmes réutilisables par le programmeur, appelés bibliothèques. La qualité et la diversité de celles-ci est souvent un des critères de facilité d'utilisation d'un langage. On peut distinguer deux catégories de bibliothèques : celles qui fournissent des types et fonctions d'usage courant définissables dans le langage et celles qui offrent des fonctionnalités non définissables dans le langage. Les premières évitent simplement au programmeur à avoir à redéfinir les utilitaires tels que les piles, les files d'attente, etc. Les secondes étendent les possibilités d'utilisation du langage en y incorporant de nouvelles fonctionnalités.

La distribution du langage Objective CAML vient avec de nombreuses bibliothèques. Elles sont fournies sous forme de fichiers compilés. Néanmoins, pour le lecteur curieux, les sources de ces bibliothèques viennent avec les distributions des sources du langage.

En Objective CAML, l'ensemble des bibliothèques est organisé en modules qui sont autant d'unités de compilation. Chacun d'entre eux contient des déclarations globales de types, d'exceptions et de valeurs qui peuvent être utilisées dans les programmes. On ne s'intéressera pas dans ce chapitre à la manière de construire de nouveaux modules, mais seulement à l'utilisation des modules existants. Le chapitre [14](#) reprendra les concepts de module (unité logique) et d'unité de compilation en décrivant le langage de modules d'Objective CAML, incluant les modules paramétrés. Pour ce qui est de la constitution de bibliothèques incorporant du code autre qu'Objective CAML, le chapitre [12](#) décrira comment intégrer aux programmes Objective CAML des traitements écrits en C.

La distribution du langage Objective CAML comporte une bibliothèque préchargée (module Pervasives), un ensemble de modules de base, appelé bibliothèque standard, et plusieurs autres bibliothèques ajoutant des fonctionnalités au langage. Certaines bibliothèques sont simplement évoquées dans ce chapitre ou décrites dans les chapitres suivants.

Plan du chapitre

Ce chapitre décrit l'ensemble des bibliothèques de la distribution du langage Objective CAML. Certaines ont déjà été rencontrées dans les chapitres précédents, comme la bibliothèque Graphics (voir chapitre 5), ou abordées comme la bibliothèque Array. La première section montre l'organisation des différentes bibliothèques. La deuxième section complète la description du module Pervasives préchargé. La troisième section classe l'ensemble des modules regroupés dans la bibliothèque standard. La quatrième section détaille les bibliothèques sur l'arithmétique exacte et sur le chargement dynamique de code.

Découpage et utilisation des bibliothèques

On découpe en trois parties l'ensemble des bibliothèques de la distribution d'Objective CAML. La première contient les déclarations globales préchargées. La deuxième, appelée bibliothèque standard, possède un bon degré de stabilité. Elle est subdivisée en quatre parties :

- structures de données
- entrées/sorties
- interface avec le système
- analyses lexicale et syntaxique

Enfin viennent les bibliothèques du troisième groupe qui apportent le plus souvent une extension au langage comme la bibliothèque Graphics (voir chapitre 5). Dans ce dernier ensemble, sont regroupées des bibliothèques portant sur les aspects suivants : expressions régulières (Str), arithmétique exacte (Num), appels au système Unix (Unix), processus légers (Threads) et chargement

dynamique de code-octet (Dynlink).

Les entrées-sorties et l'interface avec le système de la bibliothèque standard sont compatibles sur différents systèmes d'exploitation comme Unix, Windows et MacOS. Ceci n'est pas toujours le cas des bibliothèques du troisième groupe. Il existe par ailleurs de nombreuses bibliothèques distribuées de manière indépendante de la distribution Objective CAML.

Usage et nommage

Pour utiliser une déclaration globale `f` d'un module simple de nom de fichier `nom.ml` et apparaissant dans le fichier `nom.mli` s'il existe, on la qualifie par `Nom.f`. Pour éviter cela, il est possible d'ouvrir la bibliothèque pour nommer directement `f` sans le faire précéder par le nom de sa bibliothèque.

Syntaxe

open*Nom* À partir de là, toutes les déclarations globales de la bibliothèque *Nom* seront considérées comme appartenant à l'environnement global. Si deux déclarations portent le même nom dans deux bibliothèques distinctes ouvertes, alors seule la dernière déclaration est visible. Pour pouvoir appeler la première, il sera nécessaire d'utiliser la notation pointée pour faire référence à celle-ci.

Bibliothèque préchargée

Une première bibliothèque est toujours préchargée que cela soit au niveau de la boucle d'interaction ou en compilation en ligne; dans ce cas, elle est toujours liée. Elle correspond au module *Pervasives* qui est l'environnement initial du langage. Elle contient les déclarations de :

- **type** : types de base (*int*, *char*, *string*, *float*, *bool*, *unit*, *exn*, *'a array*, *'a list*) et les types *'a option* (voir page ??) et (*'a*, *'b*, *'c*) *format* (voir page ??).

- **exceptions** : une dizaine d'exceptions déclenchables par la bibliothèque d'exécution. On peut citer principalement les exceptions généralistes suivantes :
 - Failure **of** string qui est déclenchée par la fonction failwith appliquée à une chaîne.
 - Invalid_argument **of** string qui indique qu'un argument est impossible à traiter par la fonction ayant déclenché l'exception. La fonction invalid_arg appliquée à une chaîne déclenche cette exception.
 - Sys_error **of** string, pour les entrées-sorties, typiquement dans l'ouverture d'un fichier inexistant en lecture.
 - End_of_file pour la détection de fin de fichier.
 - Division_by_zero sur les entiers.

Ainsi que des exceptions plus internes comme :

- Out_of_memory et Stack_overflow pour le dépassement mémoire du tas et de la pile. Il est à noter que l'exception Out_of_memory n'est pas récupérable par un programme. En effet quand elle se déclenche, il est trop tard pour allouer de nouveau de l'espace mémoire pour poursuivre le calcul.
La récupération de l'exception Stack_Overflow diffère si le programme a été compilé en code-octet ou en natif. Dans ce dernier cas, il n'est pas possible de la rattraper.
- **fonctions** : environ 140 dont la moitié correspondent à des fonctions C de la bibliothèque d'exécution. On y trouvera les opérateurs de comparaison et de calcul, les fonctions sur les entiers et les flottants, des fonctions sur les chaînes de caractères, sur les références et les entrées-sorties. Il est à noter qu'un certain nombre de ces déclarations sont en fait des synonymes de déclarations définies dans d'autres modules. Elles sont néanmoins déclarées ici pour des raisons historiques et d'implantation.

Bibliothèque standard

La bibliothèque standard regroupe un ensemble de modules stables. Ceux-ci sont indépendants des systèmes d'exploitation. Il y a actuellement 29 modules dans la bibliothèque standard contenant 400 fonctions, 30 types dont la moitié sont abstraits, 8 exceptions, 10 sous-modules et 3 modules paramétrés. Il est clair que nous n'allons pas détailler l'ensemble des déclarations de tous ces modules. En effet le manuel de référence [[LRVD99](#)] le fait déjà très bien. Seuls les modules présentant un nouveau concept ou une réelle difficulté de manipulation seront détaillés.

La bibliothèque standard peut être découpée en quatre grandes parties :

- **structures linéaires de données** (15 modules), dont certains sont déjà apparus dans la première partie ;
- **entrées-sorties** (4 modules), pour le formatage des sorties, la persistance et la création de clés pour la cryptographie ;
- **analyses lexicale et syntaxique** (4 modules). Leur description est reportée au chapitre [11](#) (page [??](#)) ;
- **interface avec le système** qui permet de communiquer et d'analyser les paramètres passés à une commande, de naviguer dans les catalogues et d'accéder aux fichiers.

À ces quatre ensembles, on ajoute un cinquième contenant quelques utilitaires de manipulation ou de création de données comme des fonctions de traitement de caractères ou un générateur de nombres pseudo-aléatoires, etc.

Utilitaires

Les modules que nous avons baptisés « utilitaires » concernent :

- les caractères : module Char qui contient essentiellement des conversions ;
- copie d'objets : Oo qui sera présenté au chapitre [15](#) (page [??](#)), sur la programmation objet ;
- évaluation retardée : Lazy présenté en première partie, page

?? ;

- générateur de nombres aléatoires : Random que nous décrivons ci-dessous.

Génération de nombres aléatoires

Le module Random est un générateur de nombres pseudo-aléatoires. Il implante une fonction de génération de nombres aléatoires à partir d'un nombre ou d'une suite de nombres appelé racine. Pour que cette fonction ne retourne pas toujours la même suite de nombres, le programmeur doit lui passer une racine différente à chaque initialisation. À partir de cette racine la fonction engendre une suite de nombres non prévisible. Néanmoins une initialisation avec la même racine créera la même suite. Il faut donc trouver une ressource extérieure au programme, comme la date du jour en millisecondes, ou le temps passé depuis le début du programme, pour initialiser correctement le générateur.

Voici les fonctions du module :

- initialisation : `init` de type *int* -> *unit* et `full_init` de type *int array* -> *unit* qui initialisent le générateur. La deuxième fonction prend un tableau de racines.
- demande d'un nombre : `bits` de type *unit* -> *int* retourne un entier positif, `int` de type *int* -> *int* retourne un entier positif compris entre 0 et une borne passée en tant que paramètre, et `float` qui retourne un flottant en lui passant une borne.

Structures de données linéaires

Les modules sur les structures de données linéaires sont :

- modules simples : Array, String, List, Sort, Stack, Queue, Buffer, Hashtbl (qui est aussi paramétré) et Weak ;
- modules paramétrés : Hashtbl (de paramètre HashedType), Map et Set (de paramètre OrderedType).

Les modules paramétrés sont construits à partir d'autres modules accroissant ainsi leur généricité. La construction de modules paramétrés sera présentée au chapitre 14, page ??.

Structures linéaires de données simples

Le nom du module décrit le type de structures de données manipulées par le module. Si le type est abstrait, c'est-à-dire si sa représentation est masquée, la convention actuelle est de le nommer *t* à l'intérieur du module. Ces modules implantent les structures suivantes :

- module Array : vecteurs
- module List : listes
- module String : chaînes de caractères
- module Hashtbl : tables de hachage (type abstrait)
- module Buffer : chaînes de caractères extensibles (type abstrait)
- module Stack : piles (type abstrait)
- module Queue : files d'attente ou FIFO (type abstrait)
- module Weak : vecteur de pointeurs faibles (type abstrait)

Signalons un dernier module manipulant des structures linéaires :

- module Sort : tri sur les listes et les vecteurs, fusion de listes

Famille de fonctions communes

Tous ces modules, à l'exception du module Sort, définissent une structure de données, les fonctions de création et d'accès aux éléments de cette structures ainsi que des fonctions de manipulation incluant des conversions vers d'autres types. Seul le module List est sans modification physique. Nous ne donnerons pas la description complète de toutes ces fonctions. Néanmoins, nous allons nous intéresser aux familles de fonctions que l'on rencontre dans ces modules, puis nous détaillerons les modules List et Array qui sont les structures les plus communes en programmation fonctionnelle et impérative.

On retrouve peu ou prou les fonctionnalités suivantes dans tous les modules :

- une fonction `length` qui prend une valeur du type et calcule un entier correspondant à sa longueur ;
- une fonction `clear` remettant la structure linéaire à vide, si elle est modifiable ;
- une fonction d'ajout d'un élément, `add` en général, mais pouvant être dénommée différemment selon les habitudes (comme par exemple `push` pour les piles) ;
- une fonction d'accès au *i*-ème élément, souvent nommée `get` ;
- une fonction de retrait d'un élément (souvent le premier) `remove` ou `take`.

De même certaines conventions sur les fonctions de parcours et de traitement se retrouvent dans un certain nombre de modules :

- `map` : qui applique une fonction sur tous les éléments de la structure et retourne une nouvelle structure contenant les résultats de ces appels ;
- `iter` : proche de `map` mais ne tient pas compte des résultats et retourne `()`.

Pour les structures dont les éléments sont indicés. on a :

- `fill` : qui remplace (modifie en place) une partie de la structure par une valeur ;
- `blit` : qui copie une partie d'une structure dans une autre structure du même type ;
- `sub` : qui copie une partie d'une structure et en crée une nouvelle.

Modules `List` et `Array`

Nous décrivons les fonctions de ces deux bibliothèques en mettant l'accent sur les similitudes et les particularités de chacune d'elles. Pour les fonctions communes aux deux modules, *t* désigne le type '*a list*' ou '*a array*'. Lorsqu'une fonction est propre à un module, nous utiliserons la notation pointée.

Fonctionnalités communes ou analogues

La première d'entre elles est le calcul de la longueur.

`length` : `'a t -> int`

Deux fonctions permettent de concaténer deux structures ou toutes les structures d'une liste.

`append` : `'a t -> 'a t -> 'a t`

`concat` : `'a t list -> 'a t`

Chacun des deux modules possède une fonction d'accès à un élément désigné par sa position dans la structure.

`List.nth` : `'a list -> int -> 'a`

`Array.get` : `'a array -> int -> 'a`

La fonction d'accès à un élément d'indice `i` d'un vecteur `t`, dont l'usage est fréquent, possède un raccourci syntaxique : `t(i)`. Deux fonctions permettent d'appliquer un traitement (une fonction) à tous les éléments d'une structure.

`iter` : `('a -> unit) -> 'a t -> unit`

`map` : `('a -> 'b) -> 'a t -> 'b t`

On peut utiliser `iter` pour afficher le contenu d'une liste ou d'un vecteur.

```
# let print_content iter print_item xs =  
  iter (fun x -> print_string "("; print_item x; print_string  
    print_newline() ;;  
val print_content : (('a -> unit) -> 'b -> 'c) -> ('a ->  
  <fun>  
# print_content List.iter print_int [1;2;3;4;5] ;;  
(1) (2) (3) (4) (5)  
- : unit = ()  
# print_content Array.iter print_int [|1;2;3;4;5|] ;;  
(1) (2) (3) (4) (5)
```

```
- : unit = ()
```

La fonction `map` construit une nouvelle structure contenant le résultat de l'application. On peut le voir par exemple sur les vecteurs dont le contenu est modifiable :

```
# let a = [|1;2;3;4|] ;;
val a : int array = [|1; 2; 3; 4|]
# let b = Array.map succ a ;;
val b : int array = [|2; 3; 4; 5|]
# a, b;;
- : int array * int array = [|1; 2; 3; 4|], [|2; 3; 4; 5|]
```

Deux itérateurs permettent de composer l'application partielle d'une fonction à chacun des éléments d'une structure.

```
fold_left          : ('a -> 'b -> 'a) -> 'a -> 'b t -> 'a
fold_right         : ('a -> 'b -> 'b) -> 'a t -> 'b
```

Il faut fournir à ces itérateurs un cas de base qui donne la valeur par défaut lorsque la structure est vide.

```
fold_left f r [v1; v2; ...; vn] = f ... (f (f r v1) v2) ... vn
fold_right f [v1; v2; ...; vn] r = f v1 (f v2 ... (f vn r) ...)
```

Ces fonctions permettent de transformer facilement des opérations binaires en opérations n-aires. Lorsque l'opération est commutative et associative, l'itération à gauche ou à droite est indifférente :

```
# List.fold_left (+) 0 [1;2;3;4] ;;
- : int = 10
# List.fold_right (+) [1;2;3;4] 0 ;;
- : int = 10
# List.fold_left List.append [0] [[1];[2];[3];[4]] ;;
- : int list = [0; 1; 2; 3; 4]
# List.fold_right List.append [[1];[2];[3];[4]] [0] ;;
- : int list = [1; 2; 3; 4; 0]
```

Remarquons que la liste vide est élément neutre à gauche et à droite pour la concaténation binaire. On retrouve donc, pour ce cas particulier, l'équivalence des deux expressions :

```
# List.fold_left List.append [] [[1];[2];[3];[4]] ;;
- : int list = [1; 2; 3; 4]
# List.fold_right List.append [[1];[2];[3];[4]] [] ;;
- : int list = [1; 2; 3; 4]
```

On a, en fait, retrouvé la fonction `List.concat`.

Manipulations propres aux listes

Il est d'usage d'avoir sur les listes les fonctions suivantes que fournit le module `List` :

<code>List.hd</code>	:	<i>'a list -> 'a</i> premier élément de la liste
<code>List.tl</code>	:	<i>'a list -> 'a</i> liste privée de son premier élément
<code>List.rev</code>	:	<i>'a list -> 'a list</i> retournement d'une liste
<code>List.mem</code>	:	<i>'a -> 'a list -> bool</i> test d'appartenance
<code>List.flatten</code>	:	<i>'a list list -> 'a list</i> aplatissement d'une liste de listes
<code>List.rev_append</code>	:	<i>'a list -> 'a list -> 'a list</i> est égale à <i>append (rev l1) l2</i>

Les deux premières fonctions sont partielles. Elles ne sont pas définies sur la liste vide et déclenchent une exception `Failure`. Il existe une variante de `mem` : `memq` qui utilise l'égalité physique.


```
# let c = (1,2) ;;
val c : int * int = 1, 2
# let l = [c] ;;
val l : (int * int) list = [1, 2]
# List.memq (1,2) l ;;
- : bool = false
# List.memq c l ;;
- : bool = true
```

Le module List fournit deux itérateurs particuliers généralisant la conjonction et la disjonction booléennes : List.for_all et List.exists que l'on définit par itération :

```
# let for_all f xs = List.fold_right (fun x -> fun b ->
val for_all : ('a -> bool) -> 'a list -> bool = <fun>
# let exists f xs = List.fold_right (fun x -> fun b ->
val exists : ('a -> bool) -> 'a list -> bool = <fun>
```

Il existe des variantes de tous les itérateurs du module List prenant en argument deux listes et les parcourant en parallèle (iter2, map2, etc.). Si elles ne sont pas de même longueur, l'exception Invalid_argument est déclenchée. La recherche d'éléments d'une liste selon un critère donné sous forme de fonction booléenne peut se faire en utilisant les fonctions suivantes :

```
List.find          : ('a -> bool) -> 'a list -
                    > 'a
List.find_all      : ('a -> bool) -> 'a list -
                    > 'a list
```

La fonction find_all a un alias : filter. Une variante de la fonction générale de recherche est la fonction de partitionnement d'une liste :

```
List.partition     : ('a -> bool) -> 'a list -
                    > 'a list * 'a list
```

Comme utilitaires souvent nécessaires, on trouve dans le module List deux fonctions permettant de diviser ou de créer des listes de couples :

List.split	:	(<i>'a * 'b</i>) list -> 'a list * 'b list
List.combine	:	'a list -> 'b list -> (<i>'a *</i> <i>'b</i>) list

Enfin, une structure combinant listes et couples est souvent utilisée : les listes d'associations. Elles servent à stocker des valeurs associées à des clés. Ce sont des listes de couples dont la première composante joue le rôle de clé et la seconde, celui d'information associée. On a pour traiter de telles données :

List.assoc	:	'a -> (<i>'a * 'b</i>) list -> 'b extrait l'information associée à une clé
List.mem_assoc	:	'a -> (<i>'a * 'b</i>) list -> <i>bool</i> teste l'existence d'une clé
List.remove_assoc	:	'a -> (<i>'a * 'b</i>) list -> (<i>'a * 'b</i>) list suppression d'un élément correspondant à une clé

Ces fonctions ont chacune une variante utilisant l'égalité physique au lieu de l'égalité structurelle: List.assq, List.mem_assq et List.remove_assq.

Manipulations propres aux vecteurs

Les vecteurs que l'on utilise souvent en programmation impérative sont des structures physiquement modifiables. Le module Array fournit une fonction de modification de la valeur d'un élément :

Array.set	:	'a array -> int -> 'a - > unit
-----------	---	-----------------------------------

Comme get, la fonction set possède un raccourci syntaxique : *t.(i) <- a*. Il existe trois fonctions d'allocation de vecteurs :

Array.create	:	<i>int -> 'a -> 'a array</i> crée un vecteur d'une taille donnée dont tous les éléments sont initialisés avec une même valeur
Array.make	:	<i>int -> 'a -> 'a array</i> alias de create
Array.init	:	<i>int -> (int -> 'a) -> 'a array</i> crée un vecteur d'une taille donnée dont chaque élément est initialisé avec le résultat de l'application d'une fonction à l'indice de l'élément initialisé

Comme leur usage est courant, le module Array fournit deux fonctions de création de matrices (vecteurs de vecteurs) :

Array.create_matrix	:	<i>int -> int -> 'a -> 'a array array</i>
Array.make_matix	:	<i>int -> int -> 'a -> 'a array array</i>

La fonction set se généralise en une fonction modifiant les valeurs d'un intervalle décrit par un indice de départ et une longueur :

Array.fill	:	<i>'a array -> int -> int -> 'a -> unit</i>
------------	---	---

On peut copier l'intégralité d'un vecteur ou extraire un sous vecteur (décrit par un indice de début et une longueur) pour obtenir une nouvelle structure :

Array.copy	:	<i>'a array -> 'a array</i>
Array.sub	:	<i>'a array -> int -> int -> 'a array</i>

La copie ou l'extraction peuvent aussi se faire vers un autre vecteur :

```
Array.blit          :                'a array -> int -> 'a  
                                array -> int -> int ->  
                                unit
```

Le premier argument entier est l'indice dans le premier vecteur, le deuxième, l'indice dans le second vecteur et le troisième argument entier, le nombre de valeurs copiées. Les trois fonctions blit, sub et fill déclenchent l'exception Invalid_argument. L'usage privilégié des indices dans les fonctions de manipulation de vecteurs a conduit à la définition de deux itérateurs particuliers :

```
Array.iteri         :                (int -> 'a -> unit) ->  
                                'a array -> unit  
Array.mapi          :                (int -> 'a -> 'b) -> 'a  
                                array -> 'b array
```

Elles appliquent une fonction dont le premier argument est l'indice de la case traitée.

```
# let f i a = (string_of_int i) ^ ":" ^ (string_of_int a)  
  Array.mapi f [| 4; 3; 2; 1; 0 |] ;;  
- : string array = [| "0:4"; "1:3"; "2:2"; "3:1"; "4:0" |]
```

Le module Array ne fournit pas de fonction permettant de modifier en place le contenu des cases d'un vecteur avec le résultat d'un calcul sur leur contenu, mais on peut facilement l'obtenir avec iteri :

```
# let iter_and_set f t =  
  Array.iteri (fun i -> fun x -> t.(i) <- f x) t ;;  
val iter_and_set : ('a -> 'a) -> 'a array -> unit = <fun  
# let v = [|0;1;2;3;4|] ;;  
val v : int array = [|0; 1; 2; 3; 4|]  
# iter_and_set succ v ;;  
- : unit = ()  
# v ;;  
- : int array = [|1; 2; 3; 4; 5|]
```

Enfin, le module Array fournit deux fonctions de conversion avec

les listes :

Array.of_list	:	'a list -> 'a array
Array.to_list	:	'a array -> 'a list

Entrées-sorties

La bibliothèque standard comporte les quatre modules d'entrées-sorties :

- module Printf : pour le formatage en sortie;
- Format : pour un formatage en boîte gérant automatiquement les sauts de ligne;
- module Marshal : implantant un mécanisme de valeurs persistantes ;
- module Digest : pour créer des clés uniques.

La description du module Marshal sera donnée plus loin dans ce chapitre lorsque nous aborderons le traitement des données persistantes (voir page ??).

Module Printf

Le module Printf permet de formater l'affichage à la manière de la fonction printf de la bibliothèque du langage C. Le format d'affichage est représenté sous forme d'une chaîne de caractères, qui sera décodée selon les conventions du printf de C, c'est-à-dire en spécialisant le caractère %. C'est ce caractère suivi d'une lettre qui indiquera le type de l'argument à mettre à cet emplacement. Le format suivant "(x=%d, y=%d)" indique qu'il faudra placer deux entiers à la place des %d dans la chaîne de sortie.

Spécifications des formats

Un format définit des paramètres pour une chaîne d'affichage. Ceux-ci, de type de base : *int*, *float*, *char* et *string*, seront convertis en

chaînes et remplaceront leur occurrence dans la chaîne d'affichage. Les valeurs 77 et 43 fournies au format "(x=%d, y=%d)" engendreront la chaîne d'affichage complète "(x=77, y=43)". Les principales lettres indiquant le type de la conversion à effectuer sont données figure 8.1.

Type	Lettre	Résultat
entier	d ou i	décimal signé
	u	décimal non signé
	x	hexadécimal non signé en minuscule
	X	idem en majuscule
caractère	c	caractère
chaîne	s	chaîne
flottant	f	décimal
	e ou E	en notation avec exposant
	g ou G	idem
booléen	b	true ou false
spécial	a ou t	paramètre fonctionnel de type
		(out_channel -> 'a -> unit) -> 'a
		-> unit
		ou out_channel -> unit

Figure 8.1 : Convention de conversion

Le format permet aussi d'assurer le cadrage d'une conversion, ce qui permet l'alignement des valeurs à afficher. On peut principalement indiquer la taille en caractères de la conversion. Pour cela on intercalera entre le caractère % et le type de conversion un nombre

entier comme dans `%10d` qui indique une conversion essayant de se cadrer à droite sur dix caractères. Si la taille de la conversion dépasse cette limite, il ne sera pas tenu compte de cette limite. Un nombre négatif effectuera un cadrage à gauche. Pour les conversions de nombres flottants, il est utile de pouvoir spécifier la précision d'affichage. On intercalera alors un point suivi d'un nombre pour indiquer le nombre de caractères après la virgule comme dans `%.5f` qui indique cinq caractères à droite du point décimal.

Il existe deux lettres de format particulières : `a` et `t` qui indiquent un argument fonctionnel. Typiquement, une fonction d'affichage définie par l'utilisateur. C'est une spécificité d'Objective CAML.

Fonctions du module

Les types des cinq fonctions de ce module sont donnés figure 8.2.

<code>fprintf</code>	:	<code>out_channel -></code> <code>('a, out_channel,</code> <code>unit) format -></code> <code>'a</code>
<code>printf</code>	:	<code>('a, out_channel,</code> <code>unit) format -></code> <code>'a</code>
<code>eprintf</code>	:	<code>('a, out_channel,</code> <code>unit) format -></code> <code>'a</code>
<code>sprintf</code>	:	<code>('a, unit, string)</code> <code>format -> 'a</code>
<code>bprintf</code>	:	<code>Buffer.t -> ('a,</code> <code>Buffer.t, string)</code> <code>format -> 'a</code>

Figure 8.2 : Fonctions de formatage de Printf

La fonction `fprintf` prend un canal, un format et des arguments de type décrit dans le format. Les fonctions `printf` et `eprintf` sont des versions spécialisées sur la sortie standard et la sortie erreur. Enfin `sprintf` et `bprintf` n'affichent pas le résultat de la conversion, mais retournent la chaîne correspondante.

Voici quelques exemples simples d'utilisation des formats.

```
# Printf.printf "(x=%d, y=%d)" 34 78 ;;
(x=34, y=78)- : unit = ()
# Printf.printf "nom = %s, age = %d" "mapom" 18 ;;
nom = mapom, age = 18- : unit = ()
# let s = Printf.sprintf "%10.5f\n%10.5f\n" (-.12.24) (2.3)
val s : string = " -12.24000\n   2.30000\n"
# print_string s ;;
-12.24000
 2.30000
- : unit = ()
```

L'exemple suivant construit une fonction d'affichage d'une matrice de flottants sous un format donné.

```
# let print_mat m =
  Printf.printf "\n" ;
  for i=0 to (Array.length m)-1 do
    for j=0 to (Array.length m.(0))-1 do
      Printf.printf "%10.3f" m.(i).(j)
    done ;
    Printf.printf "\n"
  done ;;
val print_mat : float array array -> unit = <fun>
# print_mat (Array.create 4 [| 1.2; -.44.22; 35.2 |]) ;

 1.200    -44.220    35.200
 1.200    -44.220    35.200
 1.200    -44.220    35.200
 1.200    -44.220    35.200
- : unit = ()
```

Remarque sur le type *format*

La description d'un format adopte la syntaxe des chaînes de caractères, mais ce n'est pas une valeur de type *string*. Le décodage d'un format, selon les conventions précédentes, construit une valeur de type *format* où le paramètre 'a' est instancié soit à *unit* si le format ne mentionne pas de paramètre, soit par un type fonctionnel correspondant à une fonction pouvant recevoir autant d'arguments que mentionnés et retournant la valeur de type *unit*.

On peut illustrer ce traitement en appliquant partiellement la fonction `printf` à un format :

```
# let p3 =  
  Printf.printf "début\n%d est val1\n%s est val2\n%f est  
début  
val p3 : int -> string -> float -> unit = <fun>
```

On obtient ainsi une fonction qui attend trois arguments. Remarquez que le mot `début` a déjà été affiché. Un autre format aurait donné un autre type de fonction :

```
# let p2 =  
  Printf.printf "début\n%f est val1\n%s est val2\n";;  
début  
val p2 : float -> string -> unit = <fun>
```

En fournissant petit à petit ses arguments à `p3`, on obtient les affichages au fur et à mesure :

```
# let p31 = p3 45 ;;  
45 est val1  
val p31 : string -> float -> unit = <fun>  
# let p32 = p31 "hello" ;;  
hello est val2  
val p32 : float -> unit = <fun>  
# let p33 = p32 3.14 ;;  
3.140000 est val3  
val p33 : unit = ()  
# p33 ;;  
- : unit = ()
```

La dernière valeur obtenue n'affiche rien : c'est la valeur *()* du type *unit*.

On ne peut pas construire un format en utilisant des valeurs de type *string* :

```
# let f d =  
  Printf.printf (d^d);;  
Characters 27-30:  
This expression has type string but is here used with type  
('a, out_channel, unit) format
```

Le compilateur ne peut connaître la valeur de la chaîne passée en argument. Il ne peut donc pas connaître le type qui sert à instancier le paramètre *'a* du type format.

D'autre part, les chaînes de caractères sont des valeurs physiquement modifiables, il serait alors possible de modifier, par exemple, la partie *%d* par une autre lettre, modifiant ainsi dynamiquement un format d'affichage. C'est en contradiction avec la génération statique de la fonction de conversion.

Module Digest

Une fonction de hachage convertit une chaîne de caractères de taille quelconque en une chaîne de caractères de taille fixe, le plus souvent plus petite. Ces fonctions de hachage retournent une empreinte (digest) de leur entrée.

De telles fonctions sont utilisées pour la construction de tables de hachage, comme dans le module *Hashtbl*, permettant de tester rapidement si un élément appartient à une telle table grâce à un accès direct à l'empreinte. Par exemple la fonction *f_mod_n*, qui effectue la somme des codes ASCII des caractères d'une chaîne modulo *n*, est une fonction de hachage. Si on crée un tableau de dimension *n* pour ranger ces chaînes, de part l'empreinte on obtient un accès direct. Néanmoins deux chaînes peuvent retourner la même empreinte. Dans ces cas de collisions, on ajoute à la table de

hachage une extension pour stocker ces éléments. S'il y a trop de collisions, alors l'accès à la table de hachage est peu efficace. Si l'empreinte a une taille de n bits, alors la probabilité de collision entre deux chaînes différentes est de $1/2^n$.

Une fonction de hachage à sens unique possède une probabilité très faible de collision. Il est ainsi difficile, étant donnée une empreinte, de construire une chaîne possédant cette empreinte. La fonction précédente `f_mod_n` n'en est, à l'évidence, pas une. Les fonctions de hachage à sens unique permettent l'authentification d'une chaîne, que cela soit pour un texte envoyé sur Internet, un fichier, etc.

Le module `Digest` utilise l'algorithme MD5, pour Message Digest 5. Il retourne une empreinte sur 128 bits. Bien que l'algorithme soit public, il est impossible (à ce jour) d'effectuer une reconstruction à partir d'une empreinte. Ce module définit le type *Digest.t* comme une abréviation du type *string*. La figure 8.3 détaille les principales fonctions de ce module.

<code>string</code>	:	<i>string</i> -> <i>t</i> retourne l'empreinte d'une chaîne
<code>file</code>	:	<i>string</i> -> <i>t</i> retourne l'empreinte d'un fichier

Figure 8.3 : Fonctions du module `Digest`

On utilise la fonction `string` dans l'exemple suivant sur une petite chaîne et sur une grande construite à partir de la première. L'empreinte est toujours de taille fixe.

```
# let s = "Le petit chat est mort...";;  
val s : string = "Le petit chat est mort..."  
# Digest.string s;;
```

```
- : Digest.t = "[\138\236]5\015\156'\243\232>\176IdF\022

# let r = ref s in
  for i=1 to 100 do r:= s^ !r done;
  Digest.string !r;;
- : Digest.t = "\148\022\201?j\141\159\143\168\241;\004\
```

La création d'une empreinte sur des programmes permet de garantir le contenu de celui-ci évitant ainsi l'utilisation d'une mauvaise version. Par exemple, lors du chargement dynamique de code (voir page ??), une empreinte est utilisée pour sélectionner le fichier de code-octet à charger.

```
# Digest.file "basic.ml" ;;
- : Digest.t = "\179\026\191\137\157Ly|^w7\183\164:\167q
```

Persistence

La persistance est la conservation d'une valeur en dehors de l'exécution courante d'un programme. C'est le cas quand on écrit une valeur dans un fichier. Cette valeur est alors accessible à tout programme ayant accès au fichier. L'écriture et la lecture d'une valeur persistante réclament la définition d'un format de représentation ou codage des données. En effet, il faut savoir passer d'une structure complexe stockée en mémoire, telle un arbre, à une structure linéaire, une suite d'octets, stockée sur fichier. C'est pourquoi le codage des valeurs persistantes s'appelle linéarisation¹.

Réalisation et difficultés de la linéarisation

La mise en oeuvre d'un mécanisme de linéarisation des données demande des choix et présente des difficultés que nous décrivons ci-dessous.

- **lecture-écriture de données.** Comme la mémoire peut toujours être vue comme un vecteur de mots, une valeur peut toujours correspondre à la mémoire qu'elle occupe, quitte à

ne conserver que la partie utile en compactant alors la valeur.

- **partage ou copie.** L'aplatissement d'une donnée doit-il conserver le partage ? Typiquement un arbre binaire qui possède deux fils identiques (au sens de l'égalité physique) peut indiquer, pour le deuxième fils, qu'il a déjà sauvegardé le premier. Cette caractéristique influence la taille de la valeur sauvegardée et le temps mis pour le faire. D'autre part, en présence de valeurs physiquement modifiables, cela peut changer le comportement de cette valeur après une récupération selon que le partage a été conservé ou non.
- **structures circulaires.** Dans le cas d'une valeur circulaire, la linéarisation sans partage risque de boucler. Il sera nécessaire de conserver ce partage.
- **valeurs fonctionnelles.** Les valeurs fonctionnelles, ou fermetures, se composent d'une partie environnement et d'une partie code. La partie code correspond au point d'entrée (adresse) du code à exécuter. Que faut-il faire alors du code ? Il est possible de stocker uniquement cette adresse, mais alors seul le même programme trouvera un sens correct à cette adresse. Il est aussi possible de sauver la suite d'instructions machine de cette fonction, mais il sera nécessaire d'avoir un mécanisme de chargement dynamique de code.
- **garantie du type à la relecture.** C'est la principale difficulté de ce mécanisme. Le typage statique garantit que les valeurs typées n'engendreront pas d'erreur de type à l'exécution. Mais cela n'est vrai que pour les valeurs appartenant au programme en cours d'exécution. Quel type peut-on donner à une valeur extérieure au programme, qui n'a donc pas été vue par le vérificateur de types ? Pour seulement vérifier que la valeur relue possède le type monomorphe engendré par le compilateur, il faudrait qu'il y ait une transmission de ce type au moment de la sauvegarde, puis vérification au chargement. Avec de surcroît un mécanisme de gestion de versions des types, pour rester sûr, même en cas de redéclaration d'un type dans le programme.

Module Marshal

Le mécanisme de linéarisation du module Marshal permet, au choix,

de retenir ou non le partage des valeurs traitées. Il permet également de traiter les fermetures, mais, dans ce cas, seul le pointeur de code est conservé.

Ce module comporte principalement les fonctions de linéarisation vers un canal ou une chaîne et les fonctions de récupération à partir d'un canal ou d'une chaîne. Les fonctions de linéarisation sont paramétrables. Le type suivant déclare les deux options possibles :

```
type external_flag =  
    No_sharing  
| Closures;;
```

Le constructeur constant `No_sharing` indique de ne pas conserver le partage d'une valeur, par défaut il l'est. Le constructeur `Closures` permet de traiter les fermetures en conservant son pointeur de code. Son absence déclenchera une exception si l'on essaie de conserver une valeur fonctionnelle.

Warning

Le constructeur `Closures` est inopérant en mode interactif. Il ne peut être utilisé qu'en mode ligne de commande. Les fonctions d'écriture et de lecture de ce module sont regroupées à la figure 8.4.

<code>to_channel</code>	:	<code>out_channel -> 'a -> extern_flag list -> unit</code>
<code>to_string</code>	:	<code>'a -> extern_flag list -> string</code>
<code>to_buffer</code>	:	<code>string -> int -> int -> 'a -> extern_flag list -> unit</code>
<code>from_channel</code>	:	<code>in_channel -> 'a</code>
<code>from_string</code>	:	<code>string -> int -> 'a</code>

Figure 8.4 : Fonctions du module Marshal

La fonction `to_channel` prend un canal de sortie, une valeur et une liste d'options et écrit la valeur passée sur le canal. La fonction `to_string` produit une chaîne correspondant à la valeur linéarisée, alors que `to_buffer` effectue le même travail en modifiant une partie d'une chaîne passée en argument. La fonction `from_channel` lit sur un canal une valeur linéarisée et la retourne. La variante `from_string` prend en entrée une chaîne et la position du premier caractère à lire dans la chaîne. Plusieurs valeurs linéarisées peuvent être stockées dans le même fichier ou dans la même chaîne. Dans le premier cas, elles pourront être lues séquentiellement. Dans le deuxième cas, il faudra préciser le bon décalage par rapport au début de la chaîne pour décoder la valeur désirée.

```
# let s = Marshal.to_string [1;2;3;4] [] in String.sub s
- : string = "\132\149\166\190\000\000\000\t\000\000"
```

Warning

L'utilisation de ce module fait perdre la sûreté du typage statique (voir *infra*, page ??).

La relecture d'un objet persistant crée une valeur de type indéterminé :

```
# let x = Marshal.from_string (Marshal.to_string [1; 2;
val x : '_a = <poly>
```

Cette indétermination est marquée, en Objective CAML, par la variable de type faible `'a`. Il est recommandé de spécifier le type attendu :

```
# let l =
  let s = (Marshal.to_string [1; 2; 3; 4] []) in
    (Marshal.from_string s 0 : int list) ;;
val l : int list = [1; 2; 3; 4]
```

Nous revenons page ?? sur ce point.

Remarque

La fonction `output_value` de la bibliothèque préchargée correspond à l'appel de `to_channel` avec une liste d'options vide. La fonction `input_value` du module `Pervasives` appelle directement la fonction `from_channel`. Celles-ci ont été conservées pour la compatibilité des anciens programmes.

Exemple : sauvegarde d'écrans

On cherche à sauvegarder le bitmap, représenté comme matrice de couleurs, de toute la fenêtre graphique. La fonction `save_screen` récupère le bitmap, le convertit en tableau de couleurs et le sauve dans un fichier dont le nom a été passé en paramètre.

```
# let save_screen name =  
  let i = Graphics.get_image 0 0 (Graphics.size_x ())  
    (Graphics.size_y ()) in  
  let j = Graphics.dump_image i in  
  let oc = open_out name in  
  output_value oc j;  
  close_out oc;;  
val save_screen : string -> unit = <fun>
```

La fonction `load_screen` effectue l'opération inverse. Elle ouvre le fichier dont le nom est passé en paramètre, récupère la valeur stockée à l'intérieur, convertit cette matrice couleurs en bitmap puis affiche celui-ci.

```
# let load_screen name =  
  let ic = open_in name in  
  let dessin = ((input_value ic) : Graphics.color array a  
  close_in ic;  
  Graphics.close_graph();  
  Graphics.open_graph (":0" ^ (string_of_int (Array.length
```



```

^"x"^(string_of_int (Array.length dessin)));
let dessin2 = Graphics.make_image dessin in
Graphics.draw_image dessin2 0 0; dessin2 ;;
val load_screen : string -> Graphics.image = <fun>

```

Warning

Les valeurs dont le type est abstrait ne peuvent pas être persistantes.

C'est pour cela que l'exemple précédent n'utilise pas le type *Graphics.image* qui est abstrait, mais bien le type *color array array* qui lui est concret. L'abstraction de types est présentée au chapitre [14](#).

Partage

La perte du partage d'une donnée peut lui faire perdre complètement son intérêt. Reprenons l'exemple du générateur de symboles de la page [??](#). Pour une raison quelconque, on désire sauver les valeurs fonctionnelles `new_s` et `reset_s`, pour, par la suite, repartir de la valeur de leur compteur commun. On écrit alors le programme suivant :

```

# let reset_s,new_s =
  let c = ref 0 in
    ( function () -> c := 0 ) ,
    ( function s -> c:=!c+1; s^(string_of_int !c) ) ;;

# let save =
  Marshal.to_string (new_s,reset_s) [Marshal.Closures;Marshal.Objects]

# let (new_sl,reset_sl) =
  (Marshal.from_string save 0 : ((string -> string) * (unit -> string)))

# (* 1 *)
Printf.printf "new_s : %s\n" (new_s "X");
Printf.printf "new_s : %s\n" (new_s "X");
(* 2 *)

```

```
Printf.printf "new_s1 : %s\n" (new_s1 "X");
(* 3 *)
reset_s1();
Printf.printf "new_s1 (après reset_s1) : %s\n" (new_s1
Characters 148-154:
Unbound value new_s1
```

Les deux premiers affichages en `(* 1 *)` sont cohérents par rapport à la définition. L'affichage obtenu en `(* 2 *)` après relecture des fermetures semble également correct (après `x2` vient `x3`). Mais, en fait, le partage du compteur `c` entre les fonctions relues `new_s1` et `reset_s1` est perdu comme l'atteste l'affichage de `x4` bien que l'on ait entre temps remis le compteur à zéro. Chaque fermeture a une copie du compteur et l'appel à `reset_s1` ne remet pas à zéro le compteur de `new_s1`. Il n'aurait donc pas fallu utiliser l'option `No_sharing` lors de la linéarisation.

Il est en règle générale nécessaire de conserver le partage. Néanmoins dans certains cas où la vitesse d'exécution est importante, l'absence de partage effectue un parcours plus rapide à la sauvegarde. L'exemple suivant montre une fonction qui effectue une copie d'une matrice. Dans ce cas là aussi il peut être préférable de casser le partage :

```
# let copie_mat_f (m : float array array) =
  let s = Marshal.to_string m [Marshal.No_sharing] in
  (Marshal.from_string s 0 : float array array);;
val copie_mat_f : float array array -> float array array
```

On peut aussi l'utiliser pour la création d'une matrice sans partage :

```
# let create_mat_f n m v =
  let m = Array.create n (Array.create m v) in
  copie_mat_f m;;
val create_mat_f : int -> int -> float -> float array array
# let a = create_mat_f 3 4 3.14;;
val a : float array array =
  [| [|3.14; 3.14; 3.14; 3.14|]; [|3.14; 3.14; 3.14; 3.14|];
    [|3.14; 3.14; 3.14; 3.14|] |]
# a.(1).(2) <- 6.28;;
- : unit = ()
```

```
# a;;
- : float array array =
[| [| 3.14; 3.14; 3.14; 3.14 |]; [| 3.14; 3.14; 6.28; 3.14 |]
 [| 3.14; 3.14; 3.14; 3.14 |] |]
```

Ce qui est un comportement plus habituel que celui de `Array.create` et ressemble à celui de `Array.create_matrix`.

Taille des valeurs

Il peut être utile de connaître la taille d'un persistant. Si le partage est conservé, cette taille reflète assez bien l'occupation mémoire d'une valeur. Bien que le codage optimise parfois la taille des valeurs immédiates², l'information de taille de leur codage respectif permet de comparer différentes implantations d'une structure de données. D'autre part pour les programmes qui ne s'arrêtent jamais, logiciels embarqués ou même serveurs réseau, l'étude de la taille des données peut indiquer des fuites de mémoire. Le module `Marshal` a deux fonctions de calcul de la taille et une constante qui sont décrits à la figure 8.5.

<code>header_size</code>	:	<i>int</i>
<code>data_size</code>	:	<i>string -> int -> int</i>
<code>total_size</code>	:	<i>string -> int -> int</i>

Figure 8.5 : Fonctions de taille de `Marshal`

La taille totale d'un persistant est égale à la taille de ses données additionnée à la taille de l'en-tête.

Nous donnons ci-dessous un petit exemple d'utilisation du codage MD5 pour comparer deux représentations des arbres binaires :

```
# let taille x = Marshal.data_size (Marshal.to_string x
```

```

val taille : 'a -> int = <fun>
# type 'a bintree1 = Empty1 | Node1 of 'a * 'a bintree1
type 'a bintree1 = | Empty1 | Node1 of 'a * 'a bintree1
# let t1 =
  Node1(2, Node1(1, Node1(0, Empty1, Empty1), Empty1),
    Node1(3, Empty1, Empty1)) ;;
val t1 : int bintree1 =
  Node1
    (2, Node1 (1, Node1 (0, Empty1, Empty1), Empty1),
      Node1 (3, Empty1, Empty1))
# type 'a bintree2 =
  Empty2 | Leaf2 of 'a | Node2 of 'a * 'a bintree2 * 'a b
type 'a bintree2 =
  | Empty2
  | Leaf2 of 'a
  | Node2 of 'a * 'a bintree2 * 'a bintree2
# let t2 =
  Node2(2, Node2(1, Leaf2 0, Empty2), Leaf2 3) ;;
val t2 : int bintree2 = Node2 (2, Node2 (1, Leaf2 0, Empt
# let s1, s2 = taille t1, taille t2 ;;
val s1 : int = 13
val s2 : int = 9

```

Les valeurs données par la fonction `taille` reflètent bien l'intuition que l'on peut avoir de la taille de `t1` et `t2`.

Problème de typage

Le vrai problème avec les persistants est qu'il est possible de casser le système de types d'Objective CAML. Les fonctions de création retournent bien un type monomorphe (*unit* ou *string*). En revanche, les fonctions de récupération retournent un type polymorphe `'a`. Du point de vue des types, on pourra faire n'importe quel usage d'une valeur persistante. Voici le pire d'entre eux (voir chapitre 2, page ??) : créer une fonction `copie_magique` de type `'a -> 'b`.

```

# let copie_magique a =
  let s = Marshal.to_string a [Marshal.Closures] in

```

```
Marshal.from_string s 0;;  
val copie_magique : 'a -> 'b = <fun>
```

L'utilisation d'une telle fonction provoque un arrêt brutal de l'exécution.

```
# (copie_magique 3 : float) +. 3.1;;  
Segmentation fault
```

En mode interactif (sous Linux), on sort même de la boucle d'interaction avec un signal d'erreur système correspondant à une violation de mémoire.

Interface avec le système

La bibliothèque standard comporte six modules d'interface avec le système :

- module Sys : pour la communication entre le système d'exploitation et le programme ;
- module Arg : pour analyser les paramètres passés au programme sur la ligne de commande ;
- module Filename : pour la navigation dans les répertoires (indépendamment du système) ;
- module Printexc : pour l'interception et l'affichage des exceptions ;
- module Gc : pour le contrôle du mécanisme de récupération automatique de mémoire qui est décrit au chapitre [9](#) ;
- module Callback : pour l'appel de fonctions Objective CAML à partir de C qui est décrit au chapitre [12](#).

Les quatre premiers modules sont décrits ci-dessous.

Module Sys

Ce module apporte de bien utiles fonctions de communication avec le système d'exploitation, ainsi que le traitement des signaux reçus par un programme. Les valeurs de la figure [8.6](#) donnent des informations sur le système.

OS_type	:	<i>string</i> type du système
interactive	:	<i>bool ref</i> vrai si exécution au toplevel
word_size	:	<i>string</i> taille d'un mot (32 ou 64 bits)
max_string_length		<i>int</i> taille maximale d'une chaîne
max_array_length	:	<i>int</i> taille maximale d'un vecteur
time	:	<i>unit -> float</i> donne le temps en secondes depuis le début du programme

Figure 8.6 : Informations sur le système

La communication entre le programme et le système passe par la ligne de commande, la valeur d'une variable de l'environnement d'exécution et la possibilité de lancer un autre programme. Ces fonctions sont décrites à la figure 8.7.

argv	:	<i>string array</i> contient le vecteur de paramètres
getenv	:	<i>string -> string</i> demande de la valeur d'une variable

command	:	<i>string</i> -> <i>int</i> exécution de la commande passée en argument
---------	---	---

Figure 8.7 : Communication avec le système

Les fonctions de la figure 8.8 permettent une navigation dans la hiérarchie de fichiers.

file_exists	:	<i>string</i> -> <i>bool</i> retourne true si le fichier existe
remove	:	<i>string</i> -> <i>unit</i> destruction d'un fichier
rename	:	<i>string</i> -> <i>string</i> - > <i>unit</i> renommage d'un fichier
chdir	:	<i>string</i> -> <i>unit</i> change de catalogue courant
getcwd	:	<i>unit</i> -> <i>string</i> retourne le nom du catalogue courant

Figure 8.8 : Manipulation de fichiers

Enfin la gestion des signaux sera décrite au chapitre sur la programmation système (18).

Voici un petit programme qui reprend l'exemple sur la sauvegarde d'une fenêtre graphique sous forme d'un tableau de couleurs. La fonction `main` vérifie qu'elle n'est pas lancée de la boucle d'interaction, lit les noms des fichiers à visualiser sur la ligne de commande, teste s'ils existent et les visualise (avec la fonction `load_screen`). On intercale une attente clavier dans la fenêtre graphique entre deux visualisations.

```
# let main () =  
  if not (!Sys.interactive) then  
    for i = 0 to Array.length(Sys.argv) -1 do  
      let nom = Sys.argv.(i) in  
      if Sys.file_exists nom then  
        begin  
          ignore(load_screen nom);  
          ignore(Graphics.read_key)  
        end  
      done;;  
val main : unit -> unit = <fun>
```

Module Arg

Le module `Arg` fixe une petite syntaxe pour les arguments de la ligne de commande. Il fournit une fonction permettant l'analyse syntaxique ainsi que la définition d'une action associée aux éléments analysés. Les divers éléments de la ligne de commande sont séparés par un ou plusieurs espaces. Ils constituent les valeurs stockées dans le tableau `Sys.argv`. Dans la syntaxe fixée par `Arg`, certains éléments distingués commencent par le caractère moins (-). On les appelle des mots clés de la ligne de commande. On peut associer aux mots clés une action spécifique qui peut prendre en argument une valeur de type *string*, *int* ou *float*. La valeur de ces arguments est initialisée avec la valeur trouvée sur la ligne de commande juste après le mot clé. Il y a dans ce cas appel à une fonction de conversion des chaînes de caractères vers le type attendu. Les autres éléments de la ligne de commande sont dits arguments anonymes. On leur associe une action commune qui prend leur valeur en argument. Une option non définie provoque l'affichage d'une petite documentation en ligne de la commande

dont le contenu est défini par l'utilisateur.

Les actions associées aux mots clés sont encapsulées dans le type :

```
type spec =  
  | Unit of (unit -> unit)      (* Call the function with  
  | Set of bool ref              (* Set the reference to tr  
  | Clear of bool ref           (* Set the reference to fa  
  | String of (string -> unit) (* Call the function with  
argument *)  
  | Int of (int -> unit)         (* Call the function with  
argument *)  
  | Float of (float -> unit)    (* Call the function with  
argument *)  
  | Rest of (string -> unit)    (* Stop interpreting keywo  
function with each remaining argument*)
```

La fonction d'analyse de la ligne de commande est :

```
# Arg.parse ;;  
- : (string * Arg.spec * string) list -> (string -> unit)  
<fun>
```

Son premier argument est une liste de triplets de la forme (*key*, *spec*, *doc*) tels que :

- *key* est une chaîne de caractères correspondant à un mot clé. Elle commence donc par le caractère réservé '_'.
- *spec* est une valeur de type *spec* spécifiant l'action associée à *key*.
- *doc* est une chaîne de caractères décrivant l'option *key*. Elle est affichée sur une erreur de syntaxe.

Le deuxième argument est la fonction de traitement des arguments anonymes de la ligne de commande. Le dernier argument est une chaîne de caractères affichée en tête de la documentation en ligne.

Le module Arg comprend également :

- *Bad* : exception prenant en argument une chaîne de caractères. Elle peut être utilisée par les fonctions de

traitement.

- usage : de type *(string * Arg.spec * string) list -> string -> unit*, cette fonction affiche la documentation en ligne. On lui fournira de préférence d les mêmes arguments que ceux de parse.
- current : de type *int ref* qui contient une référence sur la valeur courante de l'indice dans le tableau Sys.argv. On peut donc modifier cette valeur si besoin est.

À titre d'exemple, nous donnons la fonction `read_args` permettant d'initialiser la configuration du démineur vu au chapitre 6, page ?? . Les options possibles seront `-col`, `-lig` et `-min`. Elles seront suivies d'un entier précisant, respectivement : le nombre de colonnes, le nombre de lignes et le nombre de mines désirées. Ces valeurs ne devront pas être inférieures aux valeurs respectives par défaut 10, 10 et 15.

Les fonctions de traitement sont :

```
# let set_nbcols cf n = cf := {!cf with nbcols = n} ;;
# let set_nbrows cf n = cf := {!cf with nbrows = n} ;;
# let set_nbmines cf n = cf := {!cf with nbmines = n} ;;
```

Elles sont toutes trois de type *config ref -> int -> unit*. La fonction d'analyse de la ligne de commande peut s'écrire :

```
# let read_args() =
  let cf = ref default_config in
  let speclist =
    [("-col", Arg.Int (set_nbcols cf), "nombre de colonnes"),
     ("lig", Arg.Int (set_nbrows cf), "nombre de lignes (>= 10)"),
     ("min", Arg.Int (set_nbmines cf), "nombre de mines (>= 15)")]
  in
  let usage_msg = "usage : demin [-col n] [-lig n] [-min n] [-h] [-v]"
  Arg.parse speclist (fun s -> ()) usage_msg; !cf ;;
val read_args : unit -> config = <fun>
```

Elle calcule une configuration qui sera passée en argument à la fonction d'ouverture de la fenêtre du jeu `open_wcf` lors du lancement du jeu. Chaque option est, comme son nom l'indique, optionnelle. Si elle ne figure pas sur la ligne de commande, le

paramètre correspondant garde la valeur par défaut. L'ordre des options n'a pas d'importance.

Module Filename

Le module `Filename` donne des opérations sur les noms de fichiers indépendantes des systèmes d'exploitation. En effet les conventions de nommage des fichiers et des catalogues diffèrent fortement entre Windows, Unix et MacOS.

Module Printexc

Ce module très court (trois fonctions décrites à la figure 8.9) apporte un récupérateur général d'exceptions. Cela est particulièrement pratique pour les programmes exécutés en mode commande³ pour être sûr de ne pas laisser s'échapper une exception qui arrêterait le programme.

<code>catch</code>	:	<code>('a -> 'b) -> 'a -</code> <code>> 'b</code> récupérateur général
<code>print</code>	:	<code>('a -> 'b) -> 'a -</code> <code>> 'b</code> affiche et redéclenche l'exception
<code>to_string</code>	:	<code>exn -> string</code> convertit une exception en chaîne

Figure 8.9 : Récupération d'exceptions

La fonction `catch` applique son premier argument à son second. Cela

lancera la fonction principale du programme. Si une exception arrive au niveau de `catch`, c'est à dire n'est pas récupérée à l'intérieur du programme, alors `catch` affichera son nom et sortira du programme. La fonction `print` a le même comportement que `catch` mais redéclenchera l'exception après l'affichage. Enfin la fonction `to_string` convertit une exception en chaîne de caractères. Elle est utilisée par les deux fonctions précédentes. Si on reprend la fonction `main` de visualisation de bitmaps, on écrira alors une fonction d'encapsulation `go` de la manière suivante :

```
# let go () =  
  Printexc.catch main ();;  
val go : unit -> unit = <fun>
```

Cela permet de terminer normalement le programme en affichant la valeur de l'exception non capturée.

Autres bibliothèques de la distribution

Les autres bibliothèques fournies avec la distribution du langage Objective CAML portent sur les extensions suivantes :

- **graphisme**, avec le module `Graphics`, portable, qui a été décrit au chapitre 5 ;
- **arithmétique exacte**, comportant plusieurs modules, et permettant d'effectuer des calculs exacts sur les entiers et les rationnels. La représentation des nombres utilise les entiers d'Objective CAML quand cela est possible ;
- **filtrage d'expressions régulières**, permettant des manipulations de chaînes et de textes plus aisées. Le module `Str` sera décrit au chapitre 11 ;
- **appels système Unix**, avec le module `Unix` qui permet d'effectuer depuis Objective CAML des appels système Unix. Une grande partie de cette bibliothèque est néanmoins compatible avec Windows. Cette bibliothèque sera utilisée aux chapitres 18 et 20 ;
- **processus légers**, comportant plusieurs modules qui seront largement décrits et utilisés au chapitre 19 ;
- **accès aux bases de données NDBD**, ne fonctionnant que sous Unix et qui ne sera pas décrit ;

- **chargement dynamique de code-octet**, constituée du seul module Dynlink.

On décrira par l'usage les bibliothèques sur les grands nombres et le chargement dynamique.

Arithmétique exacte

La bibliothèque des grands nombres offre une arithmétique exacte sur les entiers et les rationnels. Les valeurs de type *int* et *float* connaissent deux limitations : les calculs sur les entiers sont faits *modulo* le plus grand entier positif, ce qui peut provoquer des débordements passant inaperçus ; les résultats de calculs sur les flottants sont arrondis, ce qui, par propagation peut conduire à des erreurs. La bibliothèque présentée ici pallie ces défauts.

Cette bibliothèque est écrite en partie en C. Pour cette raison, il faudra construire une boucle d'interaction incluant ce code en utilisant la commande :

```
ocamlmktop -custom -o top nums.cma -cclib -lnums
```

Elle est constituée de plusieurs modules. Les deux plus importants sont Num pour toutes les opérations et Arith_status pour le contrôle des options des calculs. Le type général *num* est un type somme regroupant trois types de base :

```
type num = Int of int  
| Big_int of big_int  
| Ratio of ratio
```

Les types *big int* et *ratio* sont abstraits.

Les opérations sur les valeurs de type *num* sont suivies du symbole /. Par exemple l'addition de deux *num* s'écrira *+ /* et sera de type *num -> num -> num*. Il en sera de même pour les comparaisons. Voici un premier exemple qui calcule la factorielle :

```
# let rec fact_num n =
```

```

if Num.(<=/) n (Num.Int 0) then (Num.Int 1)
else Num.( */ ) n (fact_num ( Num.(-/) n (Num.Int 1)))
val fact_num : Num.num -> Num.num = <fun>
# let r = fact_num (Num.Int 100);;
val r : Num.num = Num.Big_int <abstr>
# let n = Num.string_of_num r in (String.sub n 0 50) ^ "
- : string = "933262154439441526816992388562667004907159

```

L'ouverture du module Num rend le code de fact_num plus facile à lire :

```

# open Num ;;
# let rec fact_num n =
  if n <=/ (Int 0) then (Int 1)
  else n */ (fact_num ( n -/ (Int 1))) ;;
val fact_num : Num.num -> Num.num = <fun>

```

Les calculs sur les rationnels sont eux aussi exacts. Si on cherche à calculer le nombre e en suivant la définition suivante :

$$e = \lim_{n \rightarrow \infty} \left(1 + \frac{1}{n} \right)^n$$

On écrira une fonction qui calcule cette limite jusqu'à un certain m .

```

# let calc_e m =
  let a = Num.(+/) (Num.Int 1) ( Num.(//) (Num.Int 1) m)
  Num.( */ ) a m;;
val calc_e : Num.num -> Num.num = <fun>
# let r = calc_e (Num.Int 100);;
val r : Num.num = Ratio <abstr>
# let n = Num.string_of_num r in (String.sub n 0 50) ^ "
- : string = "270481382942152609326719471080753083367793

```

Le module Arith_status permet de contrôler une partie des calculs comme la normalisation des rationnels, l'approximation pour affichage et le traitement des dénominateurs nuls. La fonction arith_status affiche l'état de ces indicateurs.

```
# Arith_status.arith_status();;
```

Normalization during computation --> OFF

(returned by `get_normalize_ratio ()`)

(modifiable with `set_normalize_ratio <your choice>`)

Normalization when printing --> ON

(returned by `get_normalize_ratio_when_printing ()`)

(modifiable with `set_normalize_ratio_when_printing <your choice>`)

Floating point approximation when printing rational numbers

(returned by `get_approx_printing ()`)

(modifiable with `set_approx_printing <your choice>`)

Error when a rational denominator is null --> ON

(returned by `get_error_when_null_denominator ()`)

(modifiable with `set_error_when_null_denominator <your choice>`)

```
- : unit = ()
```

Ils pourront être modifiés selon les besoins d'un calcul. Par exemple si on demande l'affichage de la valeur approchée des rationnels, on obtiendra, pour le calcul précédent :

```
# Arith_status.set_approx_printing true;;
```

```
- : unit = ()
```

```
# Num.string_of_num (calc_e (Num.Int 100));;
```

```
- : string = "0.270481382942e1"
```

Les calculs avec les grands nombres sont plus longs qu'avec les entiers et les valeurs occupent plus d'espace mémoire. Néanmoins, cette bibliothèque essaie de conserver les représentations les plus économiques. De toute façon, éviter la propagation d'erreur d'arrondis et pouvoir calculer sur des grands nombres, justifient une perte d'efficacité.

Chargement dynamique de code

Le module Dynlink offre la possibilité de charger dynamiquement des programmes sous la forme de code-octet. Le chargement dynamique de code procure les avantages suivants :

- réduire la taille en code d'un programme. Si certains modules ne sont pas utilisés, ils ne seront pas chargés.
- pouvoir choisir à l'exécution le module à charger. Selon certaines conditions d'exécution on choisit un module plutôt qu'un autre à charger.
- pouvoir modifier le comportement d'un module en cours d'exécution. Là aussi selon certaines conditions le programme peut charger un nouveau module et masquer le code du précédent.

La boucle d'interaction d'Objective CAML utilise déjà un tel mécanisme. Il est pratique que le programmeur puisse y avoir accès.

Lors du chargement d'un fichier objet (d'extension .cmo), les différentes expressions sont évaluées. Le programme principal, qui a demandé le chargement dynamique du code, n'a pas accès aux noms des déclarations. Donc c'est au module chargé dynamiquement de mettre à jour une table de fonctions utilisées par le programme principal.

Warning

Le chargement dynamique de code ne fonctionne que pour les fichiers objets en code-octet.

Description du module

Le chargement dynamique d'un fichier de code-octet f.cmo nécessite d'une part de connaître les chemins de recherche où aller trouver ce fichier et d'autre part les noms des modules dont il se sert. Par défaut, les fichiers de code-octet chargés dynamiquement n'ont pas accès aux chemins et aux modules des bibliothèques de la distribution. Il faut alors ajouter ce chemin et le nom des modules nécessaires au chargement dynamique d'un module.

<code>init</code>	:	<code>unit -> unit</code>
		initialise le
		chargement

<code>add_interfaces</code>	:	dynamique <i>string list</i> -> <i>string list</i> -> <i>unit</i> ajoute des noms de modules et des chemins de recherche pour le chargement
<code>loadfile</code>	:	<i>string</i> -> <i>unit</i> charge un fichier code-octet
<code>clear_available_units</code>		<i>unit</i> -> <i>unit</i> met à vide les noms de modules chargeables et les chemins de recherche
<code>add_available_units</code>		(<i>string</i> * <i>Digest.t</i>) <i>list</i> -> <i>unit</i> ajoute un nom de module et une empreinte pour le chargement sans avoir besoin du fichier interface
<code>allow_unsafe_modules</code>		<i>bool</i> -> <i>unit</i> accepte de charger des fichiers contenant des déclarations external
<code>loadfile_private</code>	:	<i>string</i> -> <i>unit</i> le module chargé n'est pas accessible aux prochains modules chargés

L'empreinte
d'une interface
.cmi peut être
obtenue par la
commande
extract_crc qui se
trouve dans le
catalogue des
bibliothèques de
la distribution.

Figure 8.10 : Fonctions du module Dynlink

De nombreuses erreurs peuvent survenir lors d'une demande de chargement d'un module. Non seulement le fichier doit exister avec la bonne interface dans un des chemins de recherche, mais de plus, le code-octet doit être correct et chargeable. Ces erreurs sont regroupées dans un type *error* utilisé comme argument de l'exception Error et de la fonction error de type *error* -> *string* qui permet de convertir une erreur en description claire.

Exemple

Pour écrire un petit programme qui nous permettra d'illustrer le chargement dynamique de code-octet, on se donne trois modules :

- F qui contient la définition d'une référence sur une fonction f ;
- Mod1 et Mod2 qui modifient de façon différente la fonction référencée par F.f.

Le module F est défini par le fichier f.ml :

```
let g () =  
  print_string "Je suis la fonction 'f' par défaut\n" ; f  
let f = ref g ;;
```

Le module Mod1 est défini dans le fichier mod1.ml :

```

print_string "Le module 'Mod1' modifie la valeur de 'F.f'
let g () =
  print_string "Je suis la fonction 'f' du module 'Mod1\'
  flush stdout ;;
F.f := g ;;

```

Le module Mod2 est définit dans le fichier mod2.ml :

```

print_string "Le module 'Mod2' modifie la valeur de 'F.f'
let g () =
  print_string "Je suis la fonction 'f' du module 'Mod2\'
  flush stdout ;;
F.f := g ;;

```

On définit enfin, dans le fichier main.ml, un programme principal qui appelle la fonction référencée par F.f originale, charge le module Mod1 et réappelle F.f, puis charge le module Mod2 et appelle une dernière fois la fonction F.f :

```

let main () =
  try
    Dynlink.init () ;
    Dynlink.add_interfaces [ "Pervasives"; "F" ; "Mod1" ; "
    [ Sys.getcwd() ; "/usr/local/lib/ocaml/" ] ;
    !(F.f) () ;
    Dynlink.loadfile "mod1.cmo" ;  !(F.f) () ;
    Dynlink.loadfile "mod2.cmo" ;  !(F.f) ()
  with
    Dynlink.Error e -> print_endline (Dynlink.error_message

main () ;;

```

Le programme principal doit, outre initialiser le chargement dynamique, déclarer par un appel à Dynlink.add_interfaces les interface utilisées.

On compile l'ensemble de ces modules :

```

$ ocamlc -c f.ml
$ ocamlc -o main dynlink.cma f.cmo main.ml

```

```
$ ocamlc -c f.cmo mod1.ml
$ ocamlc -c f.cmo mod2.ml
```

Si l'on exécute le programme main, on obtient :

```
$ main
Je suis la fonction 'f' par défaut
Le module 'Mod1' modifie la valeur de 'F.f'
Je suis la fonction 'f' du module 'Mod1'
Le module 'Mod2' modifie la valeur de 'F.f'
Je suis la fonction 'f' du module 'Mod2'
```

Au chargement dynamique d'un module, son code est exécuté. C'est ce que manifeste, dans notre exemple, les affichages commençant par Le module Les éventuels effets de bord qu'il contient sont donc répercutés au niveau du programme qui a effectué ce chargement. C'est pourquoi, les différents appels à F.f appellent des fonctions différentes.

La bibliothèque Dynlink offre le mécanisme de base pour le chargement dynamique de code-octet. Il reste au programmeur à gérer des tables pour que ce chargement ait véritablement un effet.

Exercices

Résolution de systèmes linéaires

Cet exercice reprend celui de résolution de systèmes linéaires présenté comme exercice du chapitre sur la programmation impérative (3).

1. En utilisant le module Printf, écrire une fonction [affiche_systeme](#) qui aligne les colonnes du système.

```
# type vect = float array
type mat = vect array
type syst = { m:mat ; v:vect } ;;
```

```

type vect = float array
type mat = vect array
type syst = { m: mat; v: vect }

# let affiche_systeme s =
  let w = Array.length s.m.(0)
  and h = Array.length s.m in
  let c = h / 2 in
  for i = 0 to h - 1 do
    Printf.printf "| ";
    for j = 0 to w - 1 do
      Printf.printf " %8.2f " s.m.(i).(j)
    done;
    Printf.printf " |";
    if i = c then Printf.printf " * " else Printf.prin
    Printf.printf "| x_%-2d |" i;
    if i = c then Printf.printf " = " else Printf.prin
    Printf.printf "| %8.2f |\n" s.v.(i)
  done;
  Printf.printf "\n" ;;
val affiche_systeme : syst -> unit = <fun>

```

2. [Tester](#) cette fonction sur les exemples donnés page [??](#).

```

# let ax3 = { m = [| [| 10.0 ; 7.0 ; 8.1 ; 7.
; [| 7.08 ; 5.04 ; 6.0 ; 5.0 |]
; [| 8.0 ; 5.98 ; 9.89 ; 9.0 |]
; [| 6.99 ; 4.99 ; 9.0 ; 9.98 |] |] ;
v = [| 32.0 ; 23.0 ; 33.0 ; 31.0 |] } ;;
val ax3 : syst =
  {m=
    [| [| 10; 7; 8.1; 7.2 |]; [| 7.08; 5.04; 6; 5 |]; [| 8;
    ... |];
    v=... }
# affiche_systeme ax3 ;;

```

10.00	7.00	8.10	7.20			x_
7.08	5.04	6.00	5.00			x_
8.00	5.98	9.89	9.00	*		x_
6.99	4.99	9.00	9.98			x_

```
- : unit = ()
```

Recherche de nombres premiers

Le crible d'Eratosthène est un algorithme simple à programmer pour la recherche de nombres premiers dans un intervalle d'entiers donné dont la borne inférieure doit être un nombre premier. La méthode est la suivante :

1. Énumérer, dans une liste, toutes les valeurs de l'intervalle.
2. Ôter de la liste toutes les valeurs multiples du premier élément.
3. Ôter de la liste et conserver ce premier élément ;
4. Recommencer en 2. tant que la liste n'est pas vide.

Voici les étapes de réalisation d'un programme qui implante cet algorithme :

1. Écrire une fonction [interval](#) qui construit un intervalle d'entiers représenté sous la forme d'une liste.

```
# let interval order next a b =  
  let rec aux a =  
    if not (order a b) then [a] else a :: aux (next a)  
  in aux a ;;  
val interval : ('a -> 'b -> bool) -> ('a -> 'a) ->  
  <fun>
```

2. Écrire la fonction [eras](#) qui calcule les nombres premiers d'un intervalle d'entiers commençant à 2 selon l'algorithme du crible d'Eratosthène.

Écrire une fonction [era_go](#) qui prend un entier et retourne la liste de tous les nombres premiers plus petits que cet entier.

```
# let rec eras l = match l with
| [] -> []
| p::q -> p :: (eras (List.filter (fun x -> x mod p > 0) q))
val eras : int list -> int list = <fun>

# let era_go n = eras (interval (<) (fun x -> x + 1) n)
val era_go : int -> int list = <fun>
```

3. On cherche à écrire un exécutable **premiers** que l'on lancera en tapant la commande **premiers n**, où **n** est un entier. Cet exécutable affichera les nombres premiers plus petits que **n**. Pour cela il faut utiliser le module **Sys** et regarder s'il y a un paramètre passé.

```
# let usage () = Printf.printf "Usage: premiers n\n"
val usage : unit -> unit = <fun>

# let main () =
  if Array.length (Sys.argv) < 2 then usage()
  else
    let n = int_of_string Sys.argv.(1) in
    if n < 2 then usage()
    else
      let r = era_go n in
      List.iter (fun x -> Printf.printf "%d " x) r;
      Printf.printf "\n" ;;
val main : unit -> unit = <fun>
```

Affichage de bitmaps

Les bitmaps sauvegardés comme *color array array* sont volumineux. Comme les 24 bits de couleur sont rarement utilisés, il est possible alors de coder un bitmap de manière plus légère. Pour cela on

analysera le nombre de couleurs d'un bitmap. Si ce nombre est petit (par exemple inférieur à 256) on pourra coder chaque pixel sur 1 octet, représentant le numéro de la couleur dans une table des couleurs de ce bitmap.

1. Écrire une fonction [analyse_couleurs](#) explorant une valeur de type *color array array* et qui retourne la liste de toutes les couleurs rencontrées dans ce tableau de couleurs.

```
# let analyse_couleurs c =  
  let l = ref [] in  
  for i = 0 to (Array.length c) - 1 do  
    for j = 0 to (Array.length c.(i)) - 1 do  
      if not(List.mem c.(i).(j) !l) then l := c.(i).(j) :  
    done ;  
  done ;  
  List.rev !l ;;  
val analyse_couleurs : 'a array array -> 'a list =
```

2. À partir de cette liste, construire une table de correspondance des couleurs. On prendra un vecteur de couleurs. L'indice du tableau correspondra au numéro d'ordre de la couleur, son contenu à la couleur. Écrire la fonction [trouve_indice](#) qui retourne l'indice d'une valeur stockée dans un tableau.

```
# let construire_table l = Array.of_list l ;;  
val construire_table : 'a list -> 'a array = <fun>  
  
# exception Find of int;;  
exception Find of int  
  
# let trouve_indice c t =  
  let aux () =  
    for i=0 to Array.length t do  
      if c = t.(i) then raise (Find i)  
    done ;  
    raise Not_found  
  in
```



```

try aux () with Find i -> i ;;
val trouve_indice : 'a -> 'a array -> int = <fun>

```

3. À partir de cette table, écrire une fonction de conversion, [encode](#), d'un *color array array* en *string*. Chaque pixel est alors représenté par un caractère.

```

# let encode caa t =
  if Array.length t > 255 then failwith "trop de couleurs"
  else
    let h = Array.length caa
    and w = Array.length caa.(0) in
    let s = String.create (h * w) in
    let ns = ref 0 in
    for i = 0 to h-1 do
      for j = 0 to w-1 do
        let ci = trouve_indice caa.(i).(j) t in
        s.[!ns] <- char_of_int ci ;
        incr ns
      done
    done ;
    s ;;
val encode : 'a array array -> 'a array -> string =

```

4. Définir un type [image_tdc](#) comportant une table de correspondance des couleurs et un vecteur de chaînes permettant de coder de manière plus petite un bitmap (ou tableau de couleurs).

```

# type image_tdc = { tdc : Graphics.color array; im
  largeur : int; hauteur : int };;
type image_tdc =
  { tdc: Graphics.color array;
    image: string;
    largeur: int;
    hauteur: int }

```

5. Écrire la fonction `to_image_tdc` de conversion d'un *color array* vers ce type.

```
# let to_image_tdc caa =  
  let t = construire_table (analyse_couleurs caa) in  
  let s = encode caa t in  
  { tdc = t; image = s;  
    largeur = Array.length caa.(0); hauteur = Array.le  
  val to_image_tdc : Graphics.color array array -> im
```

6. Écrire la fonction `sauve_image_tdc` de sauvegarde de ces valeurs vers un fichier.

```
# let sauve_image_tdc im nom =  
  let oc = open_out nom in  
  Marshal.to_channel oc im [] ;;  
val sauve_image_tdc : 'a -> string -> unit = <fun>
```

7. Comparer la taille du fichier obtenu par rapport à la sauvegarde du tableau de couleurs équivalent.

8. Écrire la fonction `from_image_tdc` de conversion inverse.

```
# let from_image_tdc im =  
  let r = Array.create_matrix im.hauteur im.largeur  
  let ns = ref 0 in  
  for i = 0 to im.hauteur -1 do  
    for j = 0 to im.largeur -1 do  
      r.(i).(j) <- im.tdc.(int_of_char im.image.[!ns])  
      incr ns  
    done  
  done ;  
  r ;;  
val from_image_tdc : image_tdc -> Graphics.color ar
```

9. L'utiliser pour **visualiser** une image sauvegardée dans un fichier sous forme d'une valeur de type *bitmap_tdc*.

```
# let visu name =  
  let ic = open_in name in  
  let im = Marshal.from_channel ic in  
  let caa = from_image_tdc im in  
  let b = Graphics.make_image caa in  
  let size = (string_of_int (Array.length caa.(0)))  
    ^ "x" ^ (string_of_int (Array.length caa)) in  
  Graphics.open_graph (":0 " ^ size) ;  
  Graphics.draw_image b 0 0 ;  
  b ;;  
val visu : string -> Graphics.image = <fun>
```

Résumé

Ce chapitre a donné une vue d'ensemble des différentes bibliothèques d'Objective CAML présentées comme des ensembles de modules simples (ou unités de compilation). Les modules sur le formatage de sortie (Printf), sur la persistance de valeurs (Marshal), d'interface avec le système (Sys) et de récupération d'exceptions (module Printexc) ont été détaillés. Les modules concernant les analyses lexicale et syntaxique, la gestion mémoire, la programmation système et réseau et les processus légers seront présentés dans les chapitres suivants.

Pour en savoir plus

Le survol des bibliothèques de la distribution du langage a montré la richesse de l'environnement de base. Pour le module Printf rien ne vaut la lecture d'un ouvrage sur le langage C, comme [HS90]. Dans [FW00] une solution est proposée pour le typage des entrées-sorties de valeurs (module Marshal). L'algorithme MD5 du module Digest est décrit sur la page de son concepteur :

Lien: <http://theory.lcs.mit.edu/~rivest/homepage.html>

De même on trouve de nombreux articles sur l'arithmétique exacte

utilisée par la bibliothèque num sur la page de Valérie Ménissier-Morain :

Lien: <http://www-calfor.lip6.fr/~vmm/>

Il existe aussi d'autres bibliothèques que celles de la distribution, développées par l'ensemble de la communauté des programmeurs Objective CAML. La majorité d'entre elles sont listées sur le site de la « bosse du chameau » :

Lien: <http://caml.inria.fr/hump.html>

Certaines d'entre elles seront présentées et discutées dans le chapitre sur le développement d'applications (22).

Pour connaître précisément le contenu des différents modules, il ne faut pas hésiter à lire la partie de description des bibliothèques du manuel de référence [LRVD99] ou de consulter sa version en ligne au format HTML (1). Pour entrer dans le détail des implantations de ces bibliothèques, rien ne vaut la lecture des sources de ces bibliothèques accessibles avec la distribution du langage (1).

Le chapitre 14 présente le langage de modules d'Objective CAML. Celui-ci permet de construire des modules simples vus comme des unités indépendantes de compilation qui correspondent aux modules présentés dans ce chapitre.

Récupérateur automatique de mémoire

Introduction

Le modèle d'exécution d'un programme sur un processeur correspond à celui de la programmation impérative. Plus précisément, un programme est une série d'instructions dont l'exécution modifie l'état mémoire de la machine. Cette mémoire est constituée principalement des valeurs créées ou manipulées par le programme. Or, comme toute ressource informatique, la mémoire disponible pour une application est une denrée finie ; un

programme tentant d'utiliser plus de ressources que le système ne lui en attribue se retrouvera dans un état incohérent. Pour cette raison, il est nécessaire d'optimiser l'utilisation de la mémoire en réutilisant la place occupée par des valeurs qui à un instant donné ne sont plus nécessaires à la suite de l'exécution. Cette gestion mémoire a une influence importante sur l'exécution du programme et sur son efficacité.

Le fait de réserver une zone de la mémoire pour en faire un certain usage s'appelle l'allocation de mémoire. On distingue l'allocation statique qui est effectuée au chargement du programme c'est-à-dire avant son exécution proprement dite, et l'allocation dynamique qui intervient au cours même de l'exécution. Alors que la mémoire allouée statiquement le reste jusqu'à la fin du programme, les zones allouées dynamiquement sont susceptibles d'être libérées, voire d'être réaffectées durant l'exécution.

La récupération d'une zone mémoire fait courir deux risques :

- si une zone mémoire est libérée alors qu'elle contient une valeur encore utilisée, cette valeur risque d'être corrompue lorsque le programme y accédera, on parle alors de pointeurs fantômes ;
- si la localisation (l'adresse) d'une zone mémoire n'est plus connue alors celle-ci ne pourra plus être libérée avant la fin du programme. On parle alors de fuite de mémoire.

Lorsque la gestion de la mémoire est effectuée par le programmeur, elle demande beaucoup de soins à ce dernier pour éviter ces deux écueils. Et cela peut s'avérer assez ardu quand les structures de données manipulées par le programme sont complexes et en particulier quand elles partagent une même zone mémoire.

Pour décharger le programmeur de cet exercice difficile, des mécanismes de récupération automatique de la mémoire ont été introduits dans de nombreux langages. L'idée maîtresse est qu'à un moment quelconque de l'exécution, les seules valeurs allouées dynamiquement susceptibles d'être utiles sont celles dont le programme connaît directement ou indirectement l'adresse. Toutes les valeurs qui ne peuvent plus être atteintes ne le pourront pas plus dans le futur donc leur zone mémoire peut être récupérée. Cette

récupération peut se faire immédiatement après qu'une valeur est devenue inutilisable, ou de manière retardée quand le programme a besoin de plus de place qu'il n'en dispose.

Objective CAML dispose d'un tel mécanisme, appelé GC (Garbage Collector en anglais et Glaneur de Cellules en français). La mémoire est allouée à la construction d'une valeur (i.e. à l'application d'un constructeur) et est libérée implicitement. Dans la plupart des programmes, il n'est besoin de connaître du GC que son existence car il joue son rôle de façon transparente. Par contre, ce mécanisme de gestion de mémoire peut avoir des conséquences sur l'efficacité du programme pour des applications créant très souvent de nouvelles valeurs. Dans de tels cas, il est utile d'avoir accès aux paramètres du GC, voire de pouvoir explicitement l'invoquer. De plus, il est nécessaire de comprendre ce qu'impose le GC à la représentation des données pour interfacer Objective CAML avec un autre langage de programmation tel que cela est décrit dans le chapitre [12](#).

Plan du chapitre

Ce chapitre présente l'allocation dynamique de mémoire et les algorithmes de récupération automatique, en particulier celui d'Objective CAML qui est une combinaison des différents algorithmes présentés. La première section rappelle les différentes classes de mémoire et leurs caractéristiques. La deuxième section décrit l'allocation mémoire dans les programmes et compare les récupérations implicites et explicites. La troisième section présente les principaux algorithmes de récupération automatique de mémoire. La quatrième section détaille celui d'Objective CAML. La cinquième section utilise le module `Gc` pour le contrôle du tas. La sixième section montre l'intérêt des *pointeurs faibles* du module `Weak` pour la création de caches mémoire.

Mémoire et programme

Un programme en langage machine est une suite d'instructions manipulant des valeurs en mémoire. Cette mémoire est généralement composée des éléments suivants:

- les registres (zone mémoire d'accès direct et rapide)
- la pile (pile d'exécution - stack)
- zone d'allocation statique (segment de données ou data segment)
- zone d'allocation dynamique (tas ou heap)

Seules les zones de la pile et d'allocation dynamique peuvent évoluer en taille pendant l'exécution d'un programme. Selon le langage de programmation utilisé, un certain contrôle de la gestion de ces éléments mémoire est ou non prévu. Dans le cadre du chargement dynamique de code (voir page ??), celui-ci prend place dans une zone d'allocation dynamique.

Allocation et récupération mémoire

La plupart des langages permettent d'allouer dynamiquement de la mémoire. On peut citer C, Pascal, Lisp, ML, SmallTalk, C + +, Java, ADA.

Allocation explicite

On distingue deux types d'allocation :

- une allocation simple qui réserve une zone mémoire d'une certaine taille sans se préoccuper de son contenu ;
- une allocation qui joint à la réservation d'un espace mémoire une initialisation pour créer des valeurs.

Le premier cas est illustré par les fonction new de Pascal ou malloc de C. Elles retournent un pointeur sur une zone mémoire (i.e. son adresse), on peut alors accéder à cette zone pour consulter ou modifier la valeur qu'elle contient. Le deuxième cas correspond aux fonctions de construction de valeurs d'Objective CAML ou celle des langages à objets. Les instances de classes des langages objets sont construites en passant à un opérateur new, la fonction de construction particulière à la classe qui, elle-même attend, en général, un certains nombre de paramètres d'initialisation. Pour les langages fonctionnels, les fonctions de constructions sont immédiatement utilisées dès que l'on définit une valeur structurée

(un couple, une liste, un enregistrement, un vecteur, une fermeture). Examinons la construction de valeurs Objective CAML sur un exemple. Leur représentation mémoire est illustrée par le schéma de la figure 9.1.

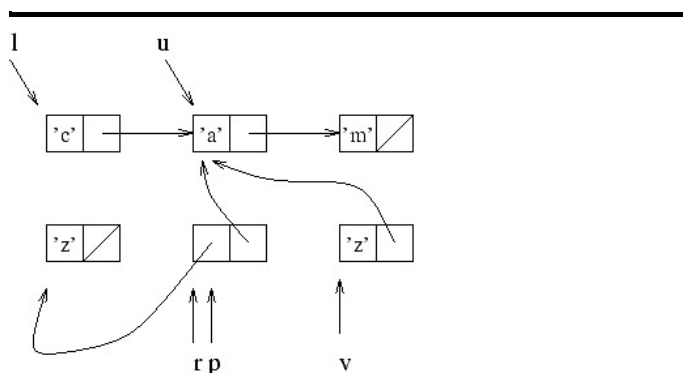


Figure 9.1 : représentation mémoire de valeurs

```
# let u = let l = ['c'; 'a'; 'm'] in List.tl l ;;
val u : char list = ['a'; 'm']
# let v = let r = ( ['z'], u )
  in match r with p -> (fst p) @ (snd p) ;;
val v : char list = ['z'; 'a'; 'm']
```

On a représenté un élément de liste par un doublet de deux mots, le premier contenant un caractère et le second un pointeur vers l'élément suivant de la liste. La représentation réelle est légèrement différente. Elle est détaillée dans le chapitre sur l'interface avec C (voir page ??).

La première phrase construit une valeur, nommée *l*, en allouant une cellule (constructeur `::`) pour chaque élément de la liste `['c'; 'a'; 'm']`. La déclaration globale *u* correspond à la queue de *l*. Il y a ici un partage entre *l* et *u*, c'est-à-dire entre l'argument et le résultat de l'application de `List.tl`. Seule la déclaration *u* est connue après l'évaluation de cette première phrase (*l* cesse d'exister).

La deuxième phrase construit une liste à un seul élément, puis un couple nommé `r` contenant cette liste et la liste `u`. Ce couple est filtré et renommé `p` par le filtrage. Ensuite, le premier élément de `p` est concaténé avec son second élément, ce qui crée une valeur `['z'; 'a'; 'm']` liée à l'identificateur global `v`. On observe que le résultat de `snd` (la liste `['a'; 'm']`) est partagé avec son argument `p` alors que le résultat de `fst` (le caractère `'z'`) est copié.

Dans tous les cas l'allocation mémoire est explicite. C'est-à-dire qu'elle est demandée par le programmeur (comme instruction ou expression du langage).

Remarque

L'allocation mémoire conserve une information sur la taille de l'objet alloué dans le but de pouvoir le libérer par la suite.

Récupération explicite

Les langages dont la récupération mémoire est explicite possèdent un opérateur de libération (`free` en C ou `dispose` en Pascal) qui reçoit l'adresse (un pointeur) de la zone à désallouer. En utilisant les informations conservées au moment de l'allocation, le programme libère cette zone et pourra la réutiliser par la suite.

L'allocation dynamique est en général utilisée pour manipuler des données structurées évolutives telles les listes, les arbres, etc. Libérer l'espace occupé par de telles données ne se fait pas en « un coup de cuillère à pot » mais réclame une fonction de parcours de la donnée. On appelle de telles fonctions des destructeurs.

Si la définition correcte des fonctions de destruction ne pose pas trop de difficulté, leur usage est, en revanche, plus délicat. En effet, pour libérer l'espace occupée par une structure, il faut en parcourir les pointeurs et appliquer l'opérateur de libération mémoire du langage. Laisser la libération de la mémoire à la charge du programmeur présente l'avantage que ce dernier est sûr des actions effectuées. Néanmoins une mauvaise utilisation de ces opérateurs peut provoquer une erreur à l'exécution du programme. Les

principaux dangers de la récupération explicite sont :

- les pointeurs « fantômes » (dangling pointers) : Une zone mémoire a été libérée alors qu'il subsiste des pointeurs qui référencent toujours cette zone. Si celle-ci est réutilisée, l'accès à cette zone par ces pointeurs risque d'être incohérent.
- les zones mémoire inaccessibles (cas de « fuite » ou leak) : Une zone mémoire toujours allouée qui n'est plus référencée par aucun pointeur. Il n'y a plus de possibilités de la libérer. Il y a une perte sèche de mémoire.

C'est toute la difficulté de la récupération explicite de mémoire que de connaître la durée de vie de l'ensemble des valeurs d'un programme.

Récupération implicite

Les langages avec récupération implicite ne possèdent pas d'opérateurs de libération mémoire. Il n'est pas possible pour le programmeur de décréter la disparition d'une valeur allouée. Néanmoins, un mécanisme de récupération automatique est déclenché quand une valeur n'est plus référencée, ou lors d'un échec de l'allocation, c'est-à-dire quand le tas est plein.

Un algorithme de récupération automatique de mémoire est en quelque sorte un destructeur global. Ce caractère rend sans doute plus difficile sa conception et son implantation que le seraient celles d'un destructeur dédié à une structure de données particulière. Mais, une fois cette difficulté surmontée, la fonction de récupération obtenue accroît considérablement la sûreté de la gestion mémoire. En particulier le risque de pointeurs fantômes disparaît.

D'autre part, une fonction de récupération automatique de mémoire peut apporter de bonnes propriétés sur le tas :

- *compaction* : toute la mémoire récupérée tient en un seul bloc, évitant ainsi la granularité du tas et permettant d'allouer des objets de la taille du tas libre ;
- *localisation* : les différents composants d'une même valeur sont

proches du point de vue adressage mémoire, ce qui permet lors de leur parcours de rester dans les mêmes pages mémoire, évitant ainsi de sortir de la mémoire-cache.

Les choix de conception d'un GC doivent tenir compte d'un certain nombre de critères et de contraintes :

- facteur de récupération : quel pourcentage de la mémoire inutilisée est disponible ?
- fragmentation de la mémoire : peut-on allouer un bloc de la taille de la mémoire libre ;
- ralentissement de l'allocation et de la récupération ;
- compilation : la représentation des valeurs est-elle totalement libre ?

En pratique, le critère de sûreté reste primordial, et les GC trouvent un compromis concernant les autres contraintes.

Récupération automatique

On distingue deux classes d'algorithmes de récupération automatique de mémoire :

- les compteurs de références : chaque zone allouée connaît le nombre de références sur elle-même. Quand ce nombre vaut zéro, la zone est libérée.
- les algorithmes explorateurs : à partir d'un ensemble de racines (pointeurs), l'ensemble des valeurs accessibles est exploré à la manière d'un parcours de graphe orienté.

Ce sont les algorithmes explorateurs qui sont le plus utilisés dans les langages de programmation. En effet, un GC à compteur de référence alourdi le traitement (mise à jour des compteurs) même lorsqu'il n'y a pas besoin de récupérer quoique ce soit.

Compteurs de références

On associe à chaque zone allouée un compteur. Ce compteur indique le nombre de pointeurs sur cet objet. Il est incrémenté à

chaque partage de l'objet. Il est décrémenté quand un pointeur sur cet objet disparaît. Quand ce compteur vaut 0, l'objet est récupéré.

L'avantage d'un tel système provient de la libération immédiate des zones qui ne sont plus utilisées. Outre l'alourdissement systématique des manipulations, les GC à compteurs de références souffrent d'un autre inconvénient : ils ne savent pas traiter les objets circulaires. Supposons qu'Objective CAML possède un tel mécanisme. L'exemple suivant construit une valeur temporaire *l*, liste de caractères dont le dernier élément pointe sur la cellule contenant 'c'. C'est donc bien une valeur circulaire (figure 9.2).

```
# let rec l = 'c'::'a'::'m'::l in List.hd l ;;  
- : char = 'c'
```

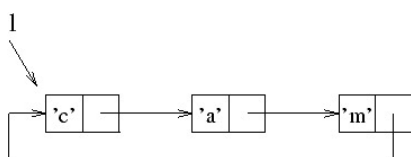


Figure 9.2 : représentation mémoire d'une liste circulaire

À la fin du calcul de cette expression, chaque élément de la liste *l* a un compteur égal à un (même le premier car la queue pointe sur la tête). Cette valeur n'est plus accessible et n'est pourtant pas récupérée car son compteur de référence n'est pas nul. Dans les langages munis d'une récupération par compteur de références, comme le langage (Python), et autorisant la construction de valeurs circulaires, il est nécessaire d'adjoindre un algorithme explorateur de mémoire.

Algorithmes explorateurs

Les algorithmes explorateurs permettent à un instant donné

d'explorer le graphe des valeurs accessibles du tas. Cette exploration utilise un ensemble de racines indiquant le début du parcours. Ces racines sont extérieures au tas. Elles sont conservées le plus souvent dans une pile. Dans notre exemple de la figure 9.1, on peut supposer que les valeurs de *u* et *v* font partie des racines. Le parcours à partir de ces racines construira le graphe des valeurs à conserver : les cellules et pointeurs en gras de la figure 9.3.

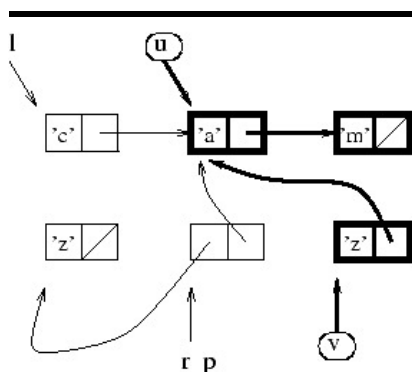


Figure 9.3 : récupération mémoire après un GC

Le parcours de ce graphe nécessite de savoir distinguer dans le tas les valeurs immédiates de pointeurs. Si une racine pointe sur un entier, il ne faut pas considérer cette valeur comme l'adresse d'une autre cellule. Dans les langages fonctionnels, cette distinction est obtenue en utilisant quelques bits des cellules du tas. On parle alors de bits d'étiquette (en anglais tag). C'est pourquoi les entiers d'Objective CAML n'utilisent que 31 bits. Cette option est décrite au chapitre 12, page ?? . Il existe d'autres solutions au problème de la distinction entre pointeurs et valeurs immédiates que nous décrivons dans ce chapitre, page ?? .

Les deux algorithmes les plus utilisés sont le *Mark&Sweep*, qui construit la liste des cellules libres du tas, et le *Stop&Copy* qui recopie les cellules encore vivantes dans une deuxième zone mémoire.

Le tas doit être vu comme un vecteur de cases mémoire. La représentation de l'état du tas de l'exemple de la figure 9.1 est illustré à la figure 9.4.

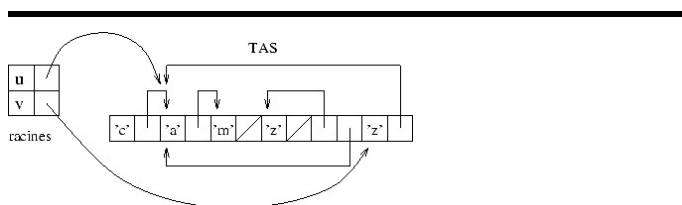


Figure 9.4 : état du tas

On évalue un algorithme explorateur par les caractéristiques suivantes.

- efficacité : la complexité en temps dépend-elle de la taille du tas ou seulement de la taille des cellules vivantes ?
- facteur de récupération : toute la mémoire libre est-elle disponible ?
- compaction : toute la mémoire libre est-elle disponible en un seul bloc ?
- localisation : les différentes cellules d'une valeur structurée sont-elles proches ?
- besoin de mémoire : l'algorithme nécessite-t'il d'utiliser une partie de la mémoire lors de son déroulement ?
- déplacement des valeurs : les valeurs changent-elles de place suite à une GC?

La localisation évite les changements de page mémoire pour le parcours d'une valeur structurée. La compaction évite la fragmentation du tas et autorise des allocations de la taille de la mémoire disponible. L'efficacité, le facteur de récupération et les besoins supplémentaires en mémoire sont intimement liés pour les complexités en temps et en espace de l'algorithme.

Le principe du *Mark&Sweep* est de tenir à jour une liste des cellules libres du tas appelée freelist. Lors d'une demande d'allocation, si la liste est vide ou s'il n'y a plus de cellules libres d'une taille suffisante alors un *Mark&Sweep* est effectué.

Il procède en deux étapes :

1. marquage des zones mémoires utiles à partir d'un ensemble de racines (phase dite *Mark*) ;
2. puis récupération des zones mémoire non marquées en parcourant séquentiellement la totalité du tas (phase dite *Sweep*).

On peut illustrer la gestion mémoire du *Mark&Sweep* en utilisant quatre « colorations » des cellules du tas : le blanc, le gris¹, le noir et le hachuré. La phase de marquage utilise le noir et le gris ; la phase de récupération utilise le hachuré et l'allocation, le blanc.

La signification des couleurs noir et gris utilisées par le marquage est la suivante :

- gris : cellule marquée dont les fils ne le sont pas encore ;
- noir : cellule marquée dont les fils immédiats le sont aussi.

Il sera nécessaire de conserver l'ensemble des cellules grisées pour être sûr de tout explorer. À la fin du marquage chaque cellule est blanche ou noire. Les cellules noires sont celles qui ont été atteintes depuis les racines. La figure 9.5 montre une étape intermédiaire du marquage pour l'exemple de la figure 9.4 : la racine u a été traitée et le traitement de v en est à son début.

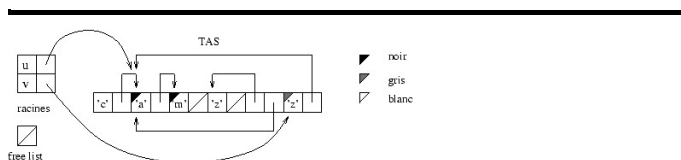


Figure 9.5 : phase de marquage

C'est dans la phase de récupération que la liste des cellules libres est construite. Elle opère les modifications suivantes de coloration :

- noir devient blanc, la cellule vit ;
- blanc devient hachuré, la cellule est ajoutée à la freelist.

La figure 9.6 montre l'évolution des couleurs du tas et la construction de la *freelist*.

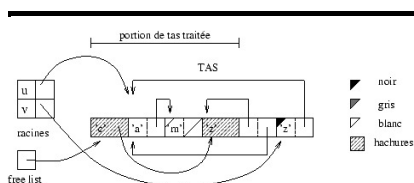


Figure 9.6 : phase de récupération

Les caractéristiques du *Mark&Sweep* sont les suivantes :

- dépend de la taille entière du tas (étape *Sweep*) ;
- récupère toute la mémoire disponible ;
- ne compacte pas la mémoire ;
- n'assure pas la localisation ;
- ne déplace pas les données .

La phase de marquage est généralement implantée par une fonction récursive. Elle utilise donc une partie de la pile d'exécution. On peut donner une version entièrement itérative de *Mark&Sweep*, n'utilisant pas une pile de profondeur inconnue, mais elle s'avère moins efficace que la version en partie récursive.

Enfin, *Mark&Sweep* a besoin de connaître la taille des valeurs. Celle-ci est soit conservée dans les valeurs, soit est déduite des adresses mémoire par un découpage du tas en zone d'allocation d'objets de taille bornée. Cet algorithme, implanté dès les premiers Lisp, est encore largement utilisé. Une partie du GC d'Objective CAML utilise un tel algorithme.

L'idée principale de ce GC est d'utiliser une mémoire secondaire pour recopier et compacter les zones mémoire à conserver. Le tas est divisé en deux parties : la zone utile (appelée from-space) et la zone de recopie (appelée to-space).

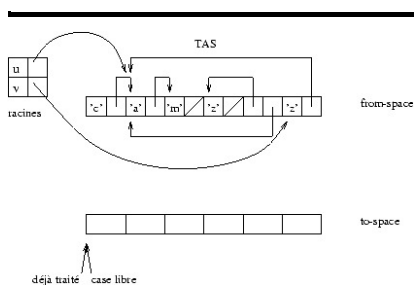


Figure 9.7 : début du *Stop&Copy*

L'algorithme est le suivant. À partir d'un ensemble de racines, on recopie la partie utile de la zone from-space dans l'espace to-space; la nouvelle adresse d'une valeur déplacée est conservée (le plus souvent dans son ancienne localisation) afin de mettre à jour toutes les autres valeurs pointant sur cette valeur.

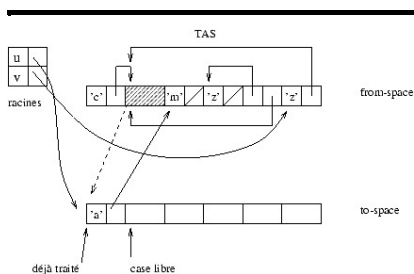


Figure 9.8 : recopie de from-space dans to-space

Le contenu des cellules recopiées forme les nouvelles racines. Tant

que toutes ne sont pas traitées l'algorithme continue.

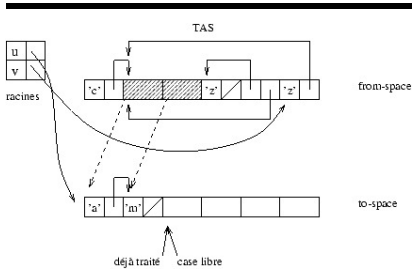


Figure 9.9 : nouvelles racines

En cas de partage, c'est-à-dire le déplacement d'une valeur déjà déplacée, on se contente de donner la nouvelle adresse.

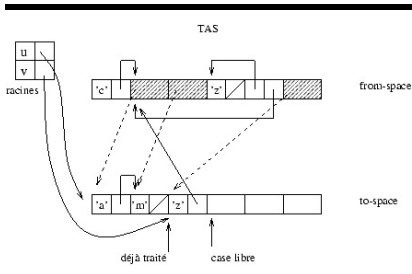


Figure 9.10 : partage

À la fin du GC, toutes les racines sont mises à jour pour pointer sur les nouvelles adresses. Enfin, les rôles des deux zones sont inversés en perspective du prochain GC.

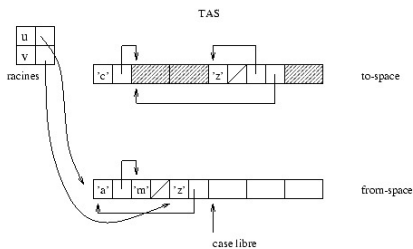


Figure 9.11 : inversion des zones

Les principales caractéristiques de ce GC sont les suivantes :

- dépend uniquement de la taille des objets à conserver;
- seule la moitié de la mémoire est disponible;
- compacte la mémoire;
- possibilité de localisation (parcours en largeur);
- n'utilise pas de mémoire supplémentaire (from-space + to-space);
- algorithme non récursif;
- déplace les données dans la nouvelle zone mémoire;

Autres GC

Bien d'autres techniques, souvent issues des deux précédentes, ont été utilisées soit dans des applications particulières, comme par exemple la manipulation de grandes matrices en calcul formel, soit de manière générale et liée à des techniques de compilation. Les GC à générations permettent d'optimiser les techniques selon l'âge des valeurs. Les GC à racines ambiguës sont utilisés quand on ne différencie pas explicitement les valeurs immédiates des pointeurs (par exemple lorsqu'on compile vers C). Enfin les GC incrémentiels permettent d'éviter un ralentissement perceptible lors du déclenchement du GC.

GC à génération

Les programmes fonctionnels sont des programmes qui allouent en général beaucoup et on constate que de très nombreuses valeurs ont une durée de vie très courte². D'autre part, dès qu'une valeur a survécu à plusieurs GC, elle a de grandes chances d'exister pour un bon moment. Pour éviter le parcours complet du tas, comme dans un *Mark&Sweep*, à chaque récupération de mémoire, on souhaite pouvoir n'effectuer un GC uniquement sur les valeurs ayant survécu à 1 ou quelques GC (c'est-à-dire des valeurs jeunes). C'est le plus souvent dans ces espaces mémoire que l'on récupérera le plus de place. Pour cela les objets sont datés, soit par une marque de temps, soit par le nombre de GC auxquels ils ont survécu. Pour optimiser le GC on utilisera des algorithmes différents selon l'âge des valeurs :

- Les GC sur les objets jeunes devront être rapides et ne parcourir que les jeunes générations.
- Les GC sur les objets anciens devront être rares et bien récupérer l'ensemble de la mémoire.

Une valeur qui vieillit sera de moins en moins partie prenante pour les GC les plus fréquents. La difficulté provient alors de ne pouvoir tenir compte uniquement de la partie mémoire des objets jeunes. Dans un langage fonctionnel pur, c'est-à-dire sans modification physique, les objets jeunes référencent des objets anciens par contre les objets anciens ne possèdent pas de pointeurs vers les objets jeunes, car ils ont été créés précédemment. Donc ces techniques s'y prêtent bien à l'exception des langages à évaluation retardée qui en fait peuvent calculer les composants d'une structure après celle-ci. Par contre pour les langages fonctionnels avec modifications physiques, il est toujours possible d'affecter une partie d'un objet ancien avec un objet jeune. Le problème est alors de conserver une zone mémoire jeune uniquement référencée par une valeur ancienne. Pour cela il sera nécessaire de tenir à jour une table des références des objets anciens vers les objets jeunes, pour obtenir un GC correct. On étudiera dans la section suivante le cas d'Objective CAML.

GC à racines ambiguës

Jusqu'à maintenant, toutes les techniques de GC supposaient savoir discriminer une valeur immédiate d'un pointeur. Il est à noter que dans les langages fonctionnels polymorphes paramétriques la représentation des valeurs est uniforme et occupe en général un mot mémoire³. C'est ce qui permet d'avoir un code générique pour les fonctions polymorphes.

Cependant, cette restriction de l'espace des entiers peut ne pas être acceptable. Dans ce cas, les GC à racines ambiguës permettent d'éviter de marquer les objets valeurs immédiates comme les entiers. Dans ce cas toutes les valeurs utilisent un mot mémoire sans aucune marque. Pour éviter le parcours d'une zone mémoire à partir d'une racine contenant un entier, on utilise un algorithme de discrimination entre valeurs immédiates et pointeurs valides reposant sur les deux observations suivantes :

- les adresses de début et de fin de tas sont connues donc toute valeur en dehors de ces bornes est une valeur immédiate;
- les objets alloués sont alignés sur une adresse de mot. Toute valeur ne correspondant pas à un alignement correct sera aussi considérée comme une valeur immédiate.

Ainsi toute valeur du tas valide du point de vue des adresses dans le tas sera comptée comme pointeur du tas et le récupérateur mémoire cherchera à conserver cette zone, y compris dans le cas où cette valeur est, en fait, une valeur immédiate. Ce cas peut devenir fort rare en utilisant des pages mémoire spécifiques selon la taille des objets. Il n'est pas possible de garantir que tout le tas inutile sera ainsi récupéré. C'est le défaut principal de cette technique. Mais on reste sûr que seules sont récupérées des zones effectivement inutilisées.

En règle générale, les GC à racines ambiguës sont conservatifs, i.e. ils ne déplacent pas les objets. En effet, puisque le GC considère certaines valeurs immédiates comme un pointeur, il serait dommageable d'en changer la valeur. Néanmoins des spécialisations peuvent être apportées dans la constitution de l'ensemble des racines et permettent de déplacer une partie des zones correspondant à des racines sûres.

Les techniques de GC à racines ambiguës sont souvent utilisées

quand on compile un langage fonctionnel vers le langage C, vu alors comme un assembleur portable. Elles permettent d'utiliser les valeurs immédiates C codées sur un mot mémoire.

GC incrémentiel

Un des reproches souvent fait aux GC est d'arrêter l'exécution du programme en cours pendant un temps perceptible à l'utilisateur et non borné. Le premier cas est gênant pour certaines applications. On peut citer les jeux d'action rapide où l'arrêt du jeu pendant quelques secondes est trop souvent source d'injustice pour le joueur, l'exécution repart sans prévenir. Le deuxième cas est source de perte de contrôle pour des applications devant traiter un certain nombre d'événements dans un temps limité. C'est typiquement le cas pour des programmes embarqués qui contrôlent un appareil physique, comme un véhicule ou une machine outil. Ces applications temps réel, dans le sens où l'application doit répondre dans un temps borné, évitent le plus souvent d'utiliser des GC.

Les GC incrémentiels doivent pouvoir s'interrompre pendant l'une quelconque de leurs phases de traitement et pouvoir repartir en assurant la sûreté de la récupération. Ils permettent de traiter de manière assez satisfaisante le premier cas et peuvent être utilisés dans le deuxième cas en forçant une discipline de programmation qui isole clairement les parties logicielles qui utilisent l'allocation dynamique de celles qui ne l'utilisent pas.

Reprenons l'exemple du *Mark&Sweep* et voyons les adaptations qu'il faut lui apporter pour le rendre incrémentiel. Il y en a essentiellement deux :

1. comment être sûr d'avoir tout marqué pendant la phase de marquage ?
2. comment allouer durant les phases de marquage ou de récupération ?

Si le *Mark&Sweep* est interrompu pendant la phase *Mark*, il faut assurer que les cellules allouées entre l'interruption du marquage et sa reprise ne seront pas indûment récupérées par le *Sweep* qui va suivre. Pour cela, on marque, par anticipation, en noir ou gris les

cellules allouées pendant l'interruption.

Si le *Mark&Sweep* est interrompu pendant la phase *Sweep*, on peut procéder de façon usuelle en remettant à blanc les marques des cellules allouées. En effet, comme la phase *Sweep* parcourt séquentiellement le tas, les cellules allouées pendant l'interruption sont localisées avant le point de redémarrage du *Sweep*, elles ne seront pas réexaminées avant le prochain cycle du GC.

La figure 9.12 montre une allocation pendant la phase de récupération. La racine *w* est créée par :

```
# let w = 'f'::v;;  
val w : char list = ['f'; 'z'; 'a'; 'm']
```

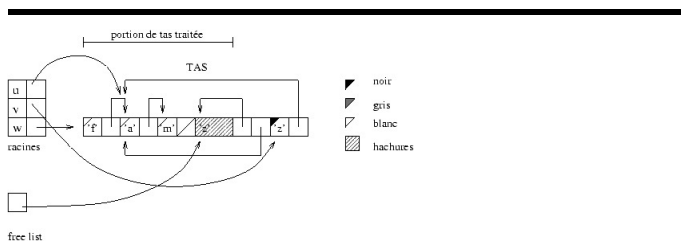


Figure 9.12 : allocation pendant la récupération

Gestion mémoire en Objective CAML

Le GC d'Objective CAML est un GC qui combine les différentes techniques rencontrées précédemment. C'est un GC à deux générations : les anciens et les nouveaux. Il utilise principalement un *Stop&Copy* sur la jeune génération (GC mineur) et un *Mark&Sweep* incrémentiel sur la génération ancienne (GC majeur).

Un objet jeune qui survit à un GC mineur est déplacé dans la génération ancienne. Le *Stop&Copy* utilise l'espace de la génération ancienne comme espace to-space. Quand il a fini, l'ensemble de

l'espace from-space est complètement libre.

Lorsque nous avons présenté les GC à génération, nous avons signalé la difficulté que présentent les langages fonctionnels impurs : une valeur de l'ancienne génération peut référencer un objet de la nouvelle génération. En voici un petit exemple :

```
# let ancien = ref [1] ;;  
val ancien : int list ref = {contents=[1]}  
(* ... *)  
# let jeune = [2;5;8] in  
  ancien := jeune ;;  
- : unit = ()
```

Le commentaire `(* ... *)` remplace une longue séquence de code pendant laquelle ancien est passé dans l'ancienne génération. Le GC mineur doit tenir compte de certaines valeurs de l'ancienne génération. On doit donc tenir à jour une table des références de l'ancienne génération vers la nouvelle qui fait partie de l'ensemble des racines dont part le GC mineur. Cette table de référence grossit très peu et elle devient vide juste après un GC mineur.

Il est à noter que le *Mark&Sweep* de la génération ancienne est incrémentiel. C'est-à-dire qu'une partie du GC majeur se déroule à chaque GC mineur. Le GC majeur est un *Mark&Sweep* qui suit l'algorithme présenté page ?? . L'intérêt de cette approche incrémentielle est de diminuer les temps d'attente du GC majeur, en avançant la phase de marquage à chaque GC mineur. Quand un GC majeur se déclenche, le marquage se termine sur les parties non traitées et la phase de récupération est enclenchée. Enfin comme le *Mark&Sweep* peut fragmenter de manière importante la génération ancienne, un algorithme de compaction peut être déclenché après le GC majeur.

On obtient au total les étapes suivantes :

1. GC mineur : effectue le *Stop&Copy* sur la génération jeune, vieillit les objets survivants en les changeant de zone et puis fait une partie du *Mark&Sweep* de la génération ancienne. Il échoue si le changement de zone échoue, on passe alors à

- l'étape 2.
2. Fin du cycle du GC majeur.
En cas d'échec on passe à l'étape 3.
 3. GC majeur de nouveau, pour vérifier si des objets comptés comme utilisés pendant les phases incrémentielles sont devenus libres.
En cas d'échec on passe à l'étape 4.
 4. Compaction de la génération ancienne pour obtenir un espace contigu maximal. En cas d'échec de cette dernière phase, il ne reste plus de possibilité et le programme échoue.

Le module GC permet de déclencher une des phases du GC.

Une dernière particularité de la gestion mémoire d'Objective CAML est que l'espace du tas n'est pas alloué une fois pour toute en début de programme, mais il évolue au cours du temps (augmentation ou diminution d'une taille donnée).

Module

Le module Gc permet d'obtenir des statistiques sur le tas et fournit un contrôle sur son évolution ainsi que sur le déclenchement des différents GC. Deux types d'enregistrements concrets sont définis : *stat* et *control*. Les champs du type *control* sont modifiables. Ils permettent d'influer sur le comportement du GC. Les champs du type *stat* ne le sont pas. Ils reflètent simplement l'état du tas à un instant donné.

Les champs d'un enregistrement *stat* contiennent principalement des compteurs indiquant :

- nombres de GC : *minor_collections*, *major_collections* et *compactions*
- nombre de mots alloués et transférés depuis le début d'exécution du programme : *minor_words*, *promoted_words*, *major_words*

Les champs d'un enregistrement *control* sont :

- *minor_heap_size* qui définit la taille de la zone dévolue à la

- jeune génération ;
- major_heap_increment qui définit l'incrément appliqué à l'accroissement de la zone de l'ancienne génération ;
- space_overhead qui définit le pourcentage de mémoire utilisé au delà duquel un GC majeur est déclenché (la valeur par défaut est 42) ;
- max_overhead qui définit le rapport entre mémoire libre et occupé à partir duquel on déclenche la compaction. La valeur 0 provoque une compaction systématique après chaque GC majeur, la valeur maximale 1000000 inhibe la compaction ;
- verbose est un paramètre entier gouvernant l'émission d'une trace de l'activité du GC.

Les fonctions manipulant les types *stat* et *control* sont données à la figure 9.13.

<i>stat</i>	<i>unit -> stat</i>
<i>print_stat</i>	<i>out_channel -> unit</i>
<i>get</i>	<i>unit @ control</i>
<i>set</i>	<i>control -> unit</i>

Figure 9.13 : fonctions de contrôle et de statistiques de tas

Les fonction suivantes de type *unit -> unit* permettent de forcer l'exécution d'un nombre d'étapes plus ou moins grand du GC d'Objective CAML : minor (étape 1), major (étapes 1 et 2), full_major (étapes 1,2 et 3) et compact (étapes 1,2,3 et 4).

Exemples

Voici ce qu'indique l'appel de GC.stat :

```
# Gc.stat();;
- : Gc.stat =
{Gc.minor_words=5828243; Gc.promoted_words=509102; Gc.ma
```

```
Gc.minor_collections=184; Gc.major_collections=9; Gc.heap_chunks=8; Gc.live_words=137371; Gc.live_blocks=137371; Gc.free_words=199502; Gc.free_blocks=1690; Gc.largest_free_block=1690; Gc.fragments=89; Gc.compactions=0}
```

On remarque le nombre d'exécutions de chaque phase : GC mineur, GC majeur, compaction, ainsi que le nombre de mots traités par les différents espaces mémoire. L'appel de compact force les quatre étapes du GC. Les informations sur le tas sont effectivement modifiées (voir appel de Gc.stat).

```
# Gc.compact();
- : unit = ()
# Gc.stat();
- : Gc.stat =
{Gc.minor_words=5834712; Gc.promoted_words=509935; Gc.major_collections=185; Gc.major_collections=11; Gc.heap_chunks=5; Gc.live_words=223345; Gc.live_blocks=223345; Gc.free_words=120718; Gc.free_blocks=2; Gc.largest_free_block=120718; Gc.fragments=1; Gc.compactions=1}
```

Les champs Gc.minor_collections et compactions sont incrémentés de 1, alors que le champ Gc.major_collections est augmenté de 2. Tous les champs du type *GC.control* sont modifiables. Pour qu'ils puissent être pris en compte, on utilisera la fonction Gc.set qui prend une valeur de type *control* et modifie le comportement du GC.

Par exemple, le champ verbose peut prendre des valeurs de 0 à 127 activant 7 indicateurs différents.

```
# c.Gc.verbose <- 31;;
Characters 1-2:
This expression has type int * int but is here used with type int
# Gc.set c;;
Characters 7-8:
This expression has type int * int but is here used with type int
# Gc.compact();
- : unit = ()
```

Ce qui affichera :

```

<>Starting new major GC cycle
allocated_words = 329
extra_heap_memory = 0u
amount of work to do = 3285u
Marking 1274 words
!Starting new major GC cycle
Compacting heap...
done.

```

Les différentes phases du GC sont indiquées ainsi que le nombre d'objets traités.

Module

Un pointeur faible (weak pointer) est un pointeur dont la zone mémoire pointée est récupérable à tout moment par le GC. Il peut être surprenant de parler d'une valeur qui peut disparaître à tout instant. En fait il faut voir ces pointeurs faibles comme un réservoir de valeurs encore disponibles. Cela s'avère particulièrement utile quand les ressources mémoire sont petites par rapport aux éléments à conserver. Le cas classique est la gestion d'un cache mémoire : une valeur peut être perdue, mais elle reste directement accessible tant qu'elle existe.

En Objective CAML on ne peut pas manipuler directement des pointeurs faibles, mais uniquement des tableaux de pointeurs faibles. Le module `Weak` définit le type abstrait `'a Weak.t` correspondant au type `'a option array`, vecteur de pointeurs faibles de type `'a`. Le type concret `'a option` est défini de la manière suivante :

```
type 'a option = None | Some of 'a;;
```

Les fonctions principales de ce module sont définies à la figure [9.14](#).

fonction	type
<code>create</code>	<code>int -> 'a t</code>
<code>set</code>	<code>'a t -> int -> 'a option -> unit</code>

get	<i>'a t -> int -> 'a option</i>
check	<i>'a t -> int -> bool</i>

Figure 9.14 : fonctions principales du module Weak

La fonction `create` alloue l'espace d'un tableau de pointeurs faibles dont chaque élément est initialisé par `None`. La fonction `set` range une valeur de type *'a option* à un indice donné. La fonction `get` retourne la valeur d'indice *n* contenue dans un tableau de pointeurs faibles. La valeur retournée est alors référencée dans l'ensemble des racines du GC et n'est plus récupérable tant que dure ce référencement. On vérifie l'existence effective d'une valeur soit en utilisant la fonction `check`, soit par filtrage sur les motifs du type *'a option*. La première solution est indépendante de la représentation des pointeurs faibles.

Les fonctions habituelles sur les structures linéaires existent aussi : `length`, pour la longueur, `fill` et `blit` pour les copies de parties de tableaux.

Exemple : cache d'images

Dans une application de construction d'images, il n'est pas rare de travailler sur plusieurs images. Quand on passe d'une image à une autre, la première est sauvegardée dans un fichier et la suivante est chargée à partir d'un autre fichier. On ne conserve en général que les noms des dernières images traitées. Pour éviter des accès au disque trop fréquents tout en n'occupant pas trop d'espace mémoire, on utilise un cache mémoire qui contient les dernières images chargées. Le contenu du cache mémoire peut être libéré si nécessaire. On l'implante par un tableau de pointeurs faibles, laissant à la charge du GC la décision de libération de ces images. Le chargement d'une image cherche tout d'abord dans le cache si l'image y est, si oui elle devient l'image courante, sinon son fichier est lu.

On définit une table des images de la manière suivante :

```
# type table_of_images = {
  size : int;
  mutable ind : int;
  mutable name : string;
  mutable current : Graphics.color array array;
  cache : ( string * Graphics.color array array) Weak.t }
```

Le champ size donne la taille du tableau ; le champ ind, l'indice de l'image courante ; le champ name, le nom de de l'image courante ; le champ current, l'image courante et le champ cache contient le tableau de pointeurs faibles sur les images. Il contient les dernières images chargées et leur nom.

La fonction init_table initialise la table avec une première image.

```
# let open_image filename =
  let ic = open_in filename
  in let i = ((input_value ic) : Graphics.color array array)
  in ( close_in ic ; i ) ;;
val open_image : string -> Graphics.color array array =

# let init_table n filename =
  let i = open_image filename
  in let c = Weak.create n
  in Weak.set c 0 (Some (filename,i)) ;
  { size=n; ind=0; name = filename; current = i; cache =
val init_table : int -> string -> table_of_images = <fun
```

Le chargement d'une nouvelle image conservera l'image courante dans la table et chargera la nouvelle. Pour ce faire, il faut tout d'abord tenter de trouver l'image dans le cache.

```
# exception Found of int * Graphics.color array array ;;
# let search_table filename table =
  try
    for i=0 to table.size-1 do
      if i<>table.ind then match Weak.get table.cache i with
        Some (n,img) when n=filename -> raise (Found (i,img))
      | _ -> ()
    done ;
```

```

None
with Found (i,img) -> Some (i,img) ;;

# let load_table filename table =
  if table.name = filename then () (* l'image est l'image courante *)
  else
    match search_table filename table with
    Some (i,img) ->
      (* l'image trouvée devient l'image courante *)
      table.current <- img ;
      table.name <- filename ;
      table.ind <- i
      | None ->
        (* l'image n'est pas dans le cache, il faut la charger *)
        (* trouver une place vide dans le cache *)
        let i = ref 0 in
        while (!i < table.size && Weak.check table.cache !i) do i := i + 1
        (* si aucune n'est libre, on en prend une pleine *)
        ( if !i = table.size then i := (table.ind + 1) mod table.size )
        (* on charge l'image à cette place et elle devient l'image courante *)
        table.current <- open_image filename ;
        table.ind <- !i ;
        table.name <- filename ;
        Weak.set table.cache table.ind (Some (filename,table.current))
val load_table : string -> table_of_images -> unit = <fun>

```

La fonction `load_table` teste si l'image demandée est l'image courante, sinon elle vérifie dans le cache si celle-ci existe, si ce n'est pas le cas elle la charge du disque; dans les deux cas elle en fait l'image courante.

Pour tester ce programme, on utilise la fonction d'affichage du cache suivante :

```

# let print_table table =
  for i = 0 to table.size-1 do
    match Weak.get table.cache ((i+table.ind) mod table.size) with
    None -> print_string "[ ] "
    | Some (n,_) -> print_string n ; print_string " "

```

```
done ;;
val print_table : table_of_images -> unit = <fun>
```

Puis on teste le programme suivant :

```
# let t = init_table 10 "IMAGES/animfond.caa" ;;
val t : table_of_images =
  {size=10; ind=0; name="IMAGES/animfond.caa";
   current=
    [| [| 7437734; 7437734; 7437734; 7437734; 7437734; 7437734;
        7437734; 7437734; 7437734; 7437734; 7437734; 7440038; 7
        ... |];
      cache=...}
# load_table "IMAGES/anim.caa" t ;;
- : unit = ()
# print_table t ;;
IMAGES/anim.caa [] [] [] [] [] [] [] [] IMAGES/animfond.
```

Cette technique de cache peut s'adapter à différentes applications.

Exercices

Suivi de l'évolution du tas

Pour suivre l'évolution du tas, on se propose d'écrire une fonction qui conserve les informations sur le tas sous la forme d'un enregistrement de la forme suivante :

```
# type tr_gc = {etat : Gc.stat;
  temps : float; numero : int};;
```

Le temps correspond au nombre de millisecondes depuis le début de l'exécution et le numéro sert à différencier les appels. On utilise la fonction `Unix.time` (voir chapitre 18, page ??) qui fournit le temps écoulé en millisecondes.

1. Écrire une fonction `trace_gc` qui retourne un tel

enregistrement.

```
# type tr_gc = {etat : Gc.stat; temps : float; num
type tr_gc = { etat: Gc.stat; temps: float; numero:
# let trace_gc =
  let num = ref 0 in
  function () -> incr num;
  { etat = Gc.stat (); temps = Unix.time(); numero
val trace_gc : unit -> tr_gc = <fun>
```

2. **Modifier** cette fonction pour qu'elle puisse sauver la valeur de type *tr_gc* dans un fichier sous forme d'une valeur persistante. Cette nouvelle fonction a besoin d'un canal en sortie pour écrire. On utilisera le module Marshal, décrit à la page ??, pour la sauvegarde des informations.

```
# let trace_out_gc oc =
  let trgc = trace_gc in
  Marshal.to_channel oc trgc [];;
val trace_out_gc : out_channel -> unit = <fun>
```

3. Écrire un **programme autonome**, prenant en entrée un nom de fichier de données de type *tr_gc* et affiche le nombre de GC mineurs et majeurs.

```
type tr_gc = {etat : Gc.stat; temps : float; numero

let rec last l = match l with
  [] -> failwith "last"
| [e] -> e
| t::q -> last q;;

let visu ltgc =
  let fst = List.hd ltgc
  and lst = last ltgc in
  let nb_minor = lst.etat.Gc.minor_collections - fst
```

```

and nb_major = lst.etat.Gc.major_collections - fst

Printf.printf "Nombre de Gc: mineur = %d, majeur =

let rec read_trace ic =
  try
    let tgc = ( Marshal.from_channel ic) : tr_gc) in
    tgc :: (read_trace ic)
  with
    End_of_file -> close_in ic; [];;

let usage () =
  Printf.printf "usage: visu filename\n";;

let main () =
  if Array.length (Sys.argv) < 2 then usage()
  else
    let ltgc = read_trace (open_in Sys.argv.(1)) in
    visu ltgc;;

main();;

```

4. Tester ce programme en créant un fichier de trace au niveau de la boucle d'interaction.

Allocation mémoire et styles de programmation

Cet exercice compare l'influence des styles de programmation sur l'évolution du tas. Pour cela on reprend l'exercice sur les nombres premiers du chapitre 8 page ?? . On cherche à comparer deux versions, une récursive terminale (voir page ??) et l'autre non, de la fonction du crible d'Eratosthène.

1. Écrire la fonction récursive terminale `erart` qui calcule les nombres entiers d'un intervalle donné. Écrire ensuite une fonction `erart_go` qui prend un entier et retourne la liste de nombres premiers inférieurs.

```
(* fichier eras2.ml *)

let erart l =
  let rec erart_aux l r = match l with
  | [] -> List.rev r
  | p::q -> erart_aux (List.filter ( fun x -> x mod p > 0 )) q
  in
  erart_aux l [] ;;

let erart_go n =
  erart (Interval.interval (<) (fun x -> x + 1) 2 n)
```

2. Écrire un `programme` qui prend un nom de fichier et une liste de nombres sur la ligne de commande et calcule pour chaque nombre la liste de nombres premiers inférieurs à celui-ci en utilisant la fonction précédente. Cette fonction effectue une trace du GC dans le fichier indiqué. On regroupe dans le fichier `trgc.ml` les commandes de trace de l'exercice précédent.

```
(* fichier : era2_main.ml *)

open Trgc;;
```

```

let usage () = Printf.printf "Usage: testgc filename\n";

let main f =
  if Array.length (Sys.argv) < 3 then usage()
  else
    let fn = Sys.argv.(1)
    and vn = Array.map int_of_string
      (Array.sub Sys.argv 2 (Array.length Sys.argv - 2))
    let oc = open_out fn in
    Array.map (fun n ->
      let r = f n in
      trace_out_gc oc;
      r) vn ;
    close_out oc ;;

(* fichier trgc.ml *)

type tr_gc = {etat : Gc.stat; temps : float; numero : int}
let trace_gc =
  let num = ref 0 in
  function () -> incr num;
  { etat = Gc.stat (); temps = Unix.time(); numero = !num }

let trace_out_gc oc =
  let trgc = trace_gc () in
  Marshal.to_channel oc trgc [];;

```

3. [Compiler](#) ces fichiers en créant un exécutable autonome et le tester sur l'appel suivant puis [afficher](#) la trace obtenue. .

```

erart trace_rt 3000 4000 5000 6000

```

```
$ ocamlc -custom -o erart unix.cma interval.ml trgc
$ erart traceRT 3000 4000 5000 6000
```

4. Effectuer le même [travail](#) pour la fonction non réursive terminale.

```
(* fichier main_nrt.ml *)

Era2_main.main Eras.era_go;;
```

5. [Comparer](#) les résultats de la trace.

Résumé

Ce chapitre a présenté les principales familles d'algorithmes de récupération automatique de mémoire dans le but de détailler celui utilisé en Objective CAML. Le GC d'Objective CAML est un GC à deux générations, incrémentiel, qui utilise un *Mark&Sweep* pour la génération ancienne et un *Stop&Copy* pour la jeune génération. Deux modules en liaison directe avec le GC permettent de contrôler l'évolution du tas. Le module Gc permet d'analyser le comportement du GC et de modifier certains paramètres dans le but de l'optimiser pour des applications spécifiques. Avec le module Weak on peut conserver dans des tableaux des valeurs potentiellement récupérables, mais qui sont encore accessibles. Il est utile pour l'implantation d'un cache mémoire.

Pour en savoir plus

Les techniques sur la récupération mémoire sont étudiées depuis quarante ans, en fait depuis les premières implantations du langage Lisp. Il y a donc une très grande littérature dans le domaine. Le seul ouvrage, en français, de présentation de ces techniques est le livre *Gestion Dynamique de la Mémoire dans les Langages de Programmation* d'Eric Spir ([[Spi90](#)]). On pourra aussi lire le tutoriel de Paul Wilson publié à IWMM92 (International Workshop on Memory Management) ([[Wil92](#)]) qui possède une importante bibliographie. De très nombreux liens permettent de connaître l'état de l'art du

domaine.

Lien: <ftp://ftp.netcom.com/pub/hb/hbaker/home.html>

est une introduction aux GC séquentiels.

Lien: <http://www.cs.ukc.ac.uk/people/staff/rej/gc.html>

contient la présentation de [Jon98] et donne accès à une imposante bibliographie.

Lien: <http://www.cs.colorado.edu/~zorn/DSA.html>

liste les différents outils de mise au point de GC.

Lien: http://reality.sgi.com/boehm_mti/

propose les sources en C d'un GC à racines ambiguës pour le langage C. Ce dernier remplace l'allocation classique malloc par une version spécialisée GC_malloc. La récupération explicite free est remplacée par une nouvelle version qui ne fait plus rien.

Lien: <http://www.harlequin.com/mm/reference/links.html>

maintient une liste de liens sur le domaine.

Le chapitre 12 sur l'interface entre les langages C et Objective CAML reviendra sur la gestion mémoire.

Outils d'analyse des programmes

Introduction

Les outils d'analyse de programmes apportent des informations complémentaires à celles données par le compilateur ou l'édition de liens. Certains de ces outils pratiquent une analyse statique. C'est-à-dire qu'ils examinent le code (sous sa forme d'un texte ou d'un arbre de syntaxe) et en déterminent certaines propriétés comme les dépendances entre modules ou les échappements d'exceptions. D'autres outils pratiquent une analyse dynamique, c'est-à-dire que

leur champ d'examen est celui de l'exécution. Ils sont utiles pour connaître par exemple le nombre d'appels aux différentes fonctions, avoir la trace du passage de leurs arguments ou savoir le temps passé dans certaines parties du programme. Ils peuvent aussi être interactifs comme le sont les outils de mise au point d'un programme. Dans ce cas l'exécution du programme est modifiée pour tenir compte de l'interaction utilisateur. Celui-ci peut poser des points d'arrêt dans le but d'explorer des valeurs ou de relancer une exécution sur de nouveaux arguments.

La distribution d'Objective CAML inclut de tels outils. Certains d'entre eux ont des caractéristiques peu communes, souvent liées au typage statique. En effet celui-ci garantit l'absence d'erreur de typage à l'exécution et est utilisé par le compilateur pour produire un code efficace et des données petites. L'information de type des valeurs Objective CAML est en partie perdue dans les valeurs construites. Cela entraîne certaines difficultés comme par exemple l'impossibilité d'afficher les arguments de fonctions polymorphes.

Plan du chapitre

Ce court chapitre présente les outils d'analyse de programme de la distribution du langage Objective CAML. La première section décrit la commande `ocamldep` qui construit les dépendances d'un ensemble de fichiers Objective CAML participant à une même application. La deuxième section s'intéresse aux outils de mise au point comme la trace des applications de fonctions ou l'outil de mise au point `ocamldebug` fonctionnant sous Unix. La troisième section regarde l'outil dit de profiling qui permet d'analyser l'exécution d'une application en vue de son optimisation.

Calcul des dépendances

Le calcul de dépendances d'un ensemble de fichiers d'implantation et d'interface d'une application Objective CAML a un double but. Le premier est d'obtenir une vision globale des interdépendances entre modules. Le deuxième est d'utiliser cette information pour ne recompiler que le strict nécessaire lors de modifications de certains fichiers.

La commande `ocamldep` prend un ensemble de fichiers `.ml` et `.mli` et affiche les dépendances entre fichiers dans le format des fichiers `Makefile`¹.

Ces dépendances proviennent de l'utilisation de déclarations globales d'autres modules, soit en utilisant la notation pointée (ex: `M1.f`), soit en ouvrant un module (ex: `open M1`).

Supposons écrits les fichiers `dp.ml` :

```
let print_vect v =
  for i = 0 to Array.length v do
    Printf.printf "%f " v.(i)
  done;
  print_newline();;
```

et `dl.ml` :

```
let init n e =
  let v = Array.create 4 3.14 in
  Dp.print_vect v;
  v;;
```

La commande `ocamldep` à qui l'on passe les noms de ces fichiers indiquera les dépendances suivantes :

```
$ ocamldep dp.ml dl.ml array.mli array.mli printf.ml prin
dp.cmo: array.cmi printf.cmi
dp.cmx: array.cmx printf.cmx
dl.cmo: array.cmi dp.cmo
dl.cmx: array.cmx dp.cmx
array.cmo: array.cmi
array.cmx: array.cmi
printf.cmo: printf.cmi
printf.cmx: printf.cmi
```

Les dépendances sont calculées pour les compilateurs code-octet et natif. On lit le résultat de la manière suivante : la production du fichier `dp.cmo` dépend des fichiers `array.cmi` et `printf.cmi`. Les fichiers d'extension `.cmi` dépendent des fichiers de même nom d'extension `.mli`. Il en est de même pour les `.ml` avec les `.cmo` et

.cmx.

Les fichiers objet de la distribution n'apparaissent pas dans la liste des dépendances. En fait le résultat précédent de `ocamldep` s'il ne trouve pas des fichiers `array.ml` et `printf.ml` dans le catalogue courant, les trouvera dans le chemin des bibliothèques de l'installation et produira :

```
$ ocamldep dp.ml dl.ml
dl.cmo: dp.cmo
dl.cmx: dp.cmx
```

Pour indiquer de nouveaux chemins de recherche de fichiers à la commande `ocamldep` on utilisera l'option de commande `-I catalogue` qui ajoute un catalogue aux chemins de recherche.

Outils de mise au point

Il y a deux outils de mise au point. Le premier outil est un mécanisme de trace utilisable, sur les fonctions globales, dans la boucle d'interaction. Le second outil est un debug sur les programmes hors boucle d'interaction. Suite à une première exécution, il permet de revenir sur des points d'arrêt, d'explorer des valeurs ou de relancer certaines fonctions avec de nouveaux paramètres. Ce second outil n'est utilisable que sous Unix car il duplique le processus programme via un `fork` (voir page ??).

Trace

On appelle trace d'une fonction l'affichage de la valeur de ses paramètres d'appels dans le cours d'un programme ainsi que celui de son résultat.

Les commandes de traçage d'exécution sont des directives de la boucle d'interaction. Elles permettent de tracer une fonction, d'arrêter sa trace ou d'arrêter toutes les traces posées. Ces trois directives sont données dans le tableau ci-dessous.

#trace nom

trace la fonction nom

#untrace nom

arrête de tracer la fonction nom

#untrace_all

annule toutes les traces

Voici un premier exemple où on définit une fonction `f` :

```
# let f x = x + 1;;  
val f : int -> int = <fun>  
# f 4;;  
- : int = 5
```

On indique maintenant que cette fonction est tracée, c'est-à-dire que le passage d'un argument ou le retour d'un résultat seront affichés.

```
# #trace f;;  
f is now traced.  
# f 4;;  
f <-- 4  
f --> 5  
- : int = 5
```

Le passage de l'argument `4` à `f` est affiché, la fonction `f` effectue le calcul demandé et le résultat retourné est aussi affiché. Les (valeurs) paramètres d'appels sont indiqués par une flèche vers la gauche et la valeur de retour, par une flèche vers la droite.

Fonctions à plusieurs arguments

Les fonctions à plusieurs arguments (ou retournant une fermeture) sont aussi traçables. Chaque passage d'un argument est affiché. Pour distinguer les différentes fermetures, on note avec une `*` le nombre d'arguments déjà passés à la fermeture. Soit la fonction `verif_div` qui prend 4 nombres (`a`, `b`, `q`, `r`) correspondant à la division entière : $a = bq + r$.

```
# let verif_div a b q r =  
  a = b*q + r;;  
val verif_div : int -> int -> int -> int -> bool = <fun>  
# verif_div 11 5 2 1;;  
- : bool = true
```

Sa trace montrera le passage des 4 arguments :

```
# #trace verif_div;;
verif_div is now traced.
# verif_div 11 5 2 1;;
verif_div <-- 11
verif_div --> <fun>
verif_div* <-- 5
verif_div* --> <fun>
verif_div** <-- 2
verif_div** --> <fun>
verif_div*** <-- 1
verif_div*** --> true
- : bool = true
```

Fonctions récursives

La trace apporte de précieuses informations sur les fonctions récursives. Elle permet de détecter aisément un mauvais critère d'arrêt.

On définit la fonction `appartient` qui teste si un entier appartient à une liste d'entiers de la manière suivante :

```
# let rec appartient (e : int) l = match l with
| [] -> false
| t::q -> (e = t) || appartient e q ;;
val appartient : int -> int list -> bool = <fun>
# appartient 4 [3;5;7] ;;
- : bool = false
# appartient 4 [1; 2; 3; 4; 5; 6; 7; 8] ;;
- : bool = true
```

La trace de l'appel de `appartient 4 [3;5;7]` affichera les quatre appels de cette fonction et les résultats retournés.

```
# #trace appartient ;;
appartient is now traced.
# appartient 4 [3;5;7] ;;
appartient <-- 4
```

```

appartient --> <fun>
appartient* <-- [3; 5; 7]
appartient <-- 4
appartient --> <fun>
appartient* <-- [5; 7]
appartient <-- 4
appartient --> <fun>
appartient* <-- [7]
appartient <-- 4
appartient --> <fun>
appartient* <-- []
appartient* --> false
appartient* --> false
appartient* --> false
appartient* --> false
- : bool = false

```

À chaque appel de la fonction `appartient` l'argument 4 puis la liste dans laquelle vérifier sont passés. Quand la liste devient vide, alors la fonction retourne la valeur `false` qui est propagée comme valeur de retour de chaque appel récursif en attente.

L'exemple suivant montre le parcours partiel de la liste lorsque l'élément demandé y apparaît :

```

# appartient 4 [1; 2; 3; 4; 5; 6; 7; 8] ;;
appartient <-- 4
appartient --> <fun>
appartient* <-- [1; 2; 3; 4; 5; 6; 7; 8]
appartient <-- 4
appartient --> <fun>
appartient* <-- [2; 3; 4; 5; 6; 7; 8]
appartient <-- 4
appartient --> <fun>
appartient* <-- [3; 4; 5; 6; 7; 8]
appartient <-- 4
appartient --> <fun>
appartient* <-- [4; 5; 6; 7; 8]
appartient* --> true
appartient* --> true

```

```
appartient* --> true
appartient* --> true
- : bool = true
```

Dès que 4 est tête de liste, la fonction retourne la valeur true qui est propagée par les appels récurifs en attente.

Si la fonction appartient avait inversé l'ordre du ||, elle retournerait toujours le bon résultat, mais effectuerait dans tous les cas le parcours complet de la liste.

```
# let rec appartient (e : int) = function
  [] -> false
  | t::q -> appartient e q || (e = t) ;;
val appartient : int -> int list -> bool = <fun>
# #trace appartient ;;
appartient is now traced.
# appartient 3 [3;5;7] ;;
appartient <-- 3
appartient --> <fun>
appartient* <-- [3; 5; 7]
appartient <-- 3
appartient --> <fun>
appartient* <-- [5; 7]
appartient <-- 3
appartient --> <fun>
appartient* <-- [7]
appartient <-- 3
appartient --> <fun>
appartient* <-- []
appartient* --> false
appartient* --> false
appartient* --> false
appartient* --> true
- : bool = true
```

Bien que 3 soit le premier élément de la liste, celle-ci est entièrement parcourue. Ainsi, le mécanisme de trace permet également une analyse d'efficacité des fonctions récursives.

Fonctions polymorphes

La trace d'arguments d'un type paramétré n'affichera pas la valeur correspondant au paramètre de type. Par exemple si la fonction `appartient` avait été écrite sans contrainte explicite de type :

```
# let rec appartient e l = match l with
| [] -> false
| t::q -> (e = t) || appartient e q ;;
val appartient : 'a -> 'a list -> bool = <fun>
```

La fonction `appartient` est maintenant d'un type polymorphe. Sa trace n'affiche plus la valeur des arguments, mais les remplace par l'indication (poly).

```
# #trace appartient ;;
appartient is now traced.
# appartient 3 [2;3;4] ;;
appartient <-- <poly>
appartient --> <fun>
appartient* <-- [<poly>; <poly>; <poly>]
appartient <-- <poly>
appartient --> <fun>
appartient* <-- [<poly>; <poly>]
appartient* --> true
appartient* --> true
- : bool = true
```

La boucle d'interaction d'Objective CAML ne sait afficher que les valeurs de type monomorphe. De plus, il ne conserve que le type inféré des déclarations globales. Ainsi, après compilation de l'expression `appartient 3 [2;3;4]`, la boucle d'interaction ne dispose plus, en fait d'information de type, que du type `'a -> 'a list -> bool` de la fonction `appartient`. Les types (monomorphes) de `3` et `[2;3;4]` ont été perdus car les valeurs ne conservent pas d'information de type : c'est le typage statique. C'est pourquoi, le mécanisme de trace attribue aux arguments de la fonction `appartient` les types polymorphes `'a` et `'a list` dont il n'affiche pas les valeurs.

C'est l'absence d'information de type dans les valeurs qui entraîne l'impossibilité de construire une fonction `print` générique de type `'a -> unit`.

Fonctions locales

Les fonctions locales ne sont pas traçables pour les mêmes raisons liées au typage statique. Seuls les types des déclarations globales sont conservés dans l'environnement de la boucle d'interaction. Pourtant le style de programmation suivant est courant :

```
# let appartient e l =  
  let rec app_aux l = match l with  
    [] -> false  
    | t::q -> (e = t) || (app_aux q)  
  in  
    app_aux l;;  
val appartient : 'a -> 'a list -> bool = <fun>
```

La fonction globale ne fait qu'appeler une fonction locale effectuant la partie intéressante du travail.

Remarques sur la trace

Le mécanisme de trace est actuellement l'unique outil de mise au point multi-plate-formes. Ses deux faiblesses sont l'absence de trace des déclarations locales et le non affichage des paramètres polymorphes des fonctions. Cela restreint fortement son usage, principalement aux premiers pas avec le langage.

Debug

L'outil de mise au point des programmes, `ocamldebug`, est un débogueur au sens usuel du terme. Il permet l'exécution pas-à-pas, l'insertion de points d'arrêt, l'exploration et la modification des valeurs de l'environnement.

Exécuter pas-à-pas un programme suppose que l'on sache déterminer ce qu'est un pas de programme. En programmation impérative, cette notion est assez simple : grosso modo, un pas de programme est une instruction du langage. Mais cette définition n'a plus grand sens en programmation fonctionnelle; on parle plutôt d'événement (event) de programmes. Ceux-ci seront une application, l'entrée dans une fonction, un filtrage, une alternative, une boucle, un élément de séquence, etc.

Warning

Cet outil ne fonctionne que sous Unix.

Compilation en mode debug

L'option de compilation `-g` engendre un fichier `.cmo` permettant d'engendrer les instructions nécessaires au debug. Cette option n'est connue que du compilateur de code-octet. Il sera nécessaire d'indiquer cette option lors de la compilation des différents fichiers d'une application. Une fois un exécutable produit, son exécution en mode debug s'effectue par la commande `ocamldebug` de la manière suivante :

On prend l'exemple suivant du fichier `fact.ml` de calcul de la fonction factorielle :

```
let fact n =  
  let rec fact_aux p q n =  
    if n = 0 then p  
    else fact_aux (p+q) p (n-1)  
  in  
  fact_aux 1 1 n;;
```

Le programme principal du fichier `main.ml` part pour une longue récursion suite à l'appel de `Fact.fact` à `-1`.

```
let x = ref 4;;  
let go () =  
  x := -1;
```



```
Fact.fact !x;;  
go();;
```

On compile les deux fichiers avec l'option -g :

```
$ ocamlc -g -i -o fact.exe fact.ml main.ml  
val fact : int -> int  
val x : int ref  
val go : unit -> int
```

Lancement du debug

Une fois un exécutable créé en mode debug, il est possible de démarrer l'exécution dans ce mode.

```
$ ocamldebug fact.exe  
Objective Caml Debugger version 2.04
```

```
(ocd)
```

Suivi du contrôle d'exécution

Le contrôle d'exécution passe par les événements du programme. On peut soit avancer ou reculer de n éléments de programme, soit avancer ou reculer jusqu'au point d'arrêt le plus proche (ou de n points d'arrêt). On indique un point d'arrêt sur une fonction ou sur un élément de programme. Le choix de l'élément de langage est donné par les coordonnées en ligne, colonne ou numéro de caractères. Ces localisations peuvent être relatives à un module.

Dans l'exemple ci-dessous, on pose un point d'arrêt sur la quatrième ligne du module Main :

```
(ocd) step 0  
Loading program... done.  
Time : 0  
Beginning of program.
```

```
(ocd) break @ Main 4
```

```
Breakpoint 1 at 4964 : file Main, line 4 column 3
```

Les initialisations du module se font avant le programme proprement dit. C'est pour cette raison que le point d'arrêt à la ligne 4 ne se rencontre qu'après 4964 instructions.

On avance ou recule dans l'exécution soit par rapport aux éléments de programme, soit par rapport aux points d'arrêt. `run` et `reverse` exécutent jusqu'au point d'arrêt le plus proche, le premier dans le sens de l'exécution, le second en marche arrière. La commande `step` avance de 1 ou n éléments de programme en entrant dans les fonctions, `next` n'entre pas dans les appels de fonctions. Respectivement `backstep` et `previous` effectuent les mêmes opérations en marche arrière. Enfin `finish` termine l'appel de la fonction courante alors que `start` retourne à l'élément de programme précédant l'invocation de la fonction.

Pour suivre notre exemple, on avance jusqu'au point d'arrêt puis on exécute trois événements de programme :

```
(ocd) run
```

```
Time : 6 - pc : 4964 - module Main
```

```
Breakpoint : 1
```

```
4 <|b|>Fact.fact !x;;
```

```
(ocd) step
```

```
Time : 7 - pc : 4860 - module Fact
```

```
2 <|b|>let rec fact_aux p q n =
```

```
(ocd) step
```

```
Time : 8 - pc : 4876 - module Fact
```

```
6 <|b|>fact_aux 1 1 n;;
```

```
(ocd) step
```

```
Time : 9 - pc : 4788 - module Fact
```

```
3 <|b|>if n = 0 then p
```

Exploration de valeurs

Au niveau d'un arrêt, il est possible d'afficher les valeurs associées aux variables du bloc d'activation. Les commandes `print` et `display` affichent selon des profondeurs différentes les valeurs associées à

une valeur.

Nous allons afficher la valeur de n, puis remonter de trois pas pour afficher le contenu de x :

```
(ocd) print n
n : int = -1
(ocd) backstep 3
Time : 6 - pc : 4964 - module Main
Breakpoint : 1
4    <|b|>Fact.fact !x;;
(ocd) print x
x : int ref = {contents=-1}
```

Les accès aux champs d'un enregistrement ou via l'indice d'un tableau sont acceptés par les commandes d'affichage.

```
(ocd) print x.contents
$1 : int = -1
```

Pile d'exécution

La pile d'exécution permet de visualiser l'imbrication des appels de fonctions. La commande backtrace ou bt montre l'empilement des appels. Les commandes up et down sélectionnent le bloc d'activation suivant ou précédant. Enfin la commande frame décrit le bloc courant.

Profiling

Cet outil permet d'obtenir un certain nombre de mesures sur l'exécution d'un programme : nombre de passages dans une fonction ou dans certaines structures de contrôle (conditionnelles, filtrages et boucles). Les informations sont enregistrées dans un fichier. De leur examen on peut déduire des erreurs algorithmiques ou les points cruciaux à optimiser dans une application.

Pour pouvoir traiter le code, il est nécessaire de compiler l'exécutable dans un mode spécial qui lui ajoute les instructions

nécessaires aux divers décomptes. Il existe deux modes de profiling, un pour le compilateur de code-octet et un autre pour le compilateur natif, ainsi que deux commandes pour la présentation des résultats. Dans l'analyse du code natif on récupère aussi le temps passé dans chaque fonction.

Le profiling d'une application se déroule donc en trois temps :

1. compilation en mode profiling;
2. exécution du programme;
3. présentation des mesures.

Commandes de compilation

Les commandes de compilation en mode profiling sont les suivantes :

- `ocamlcp -p options` pour le compilateur de code-octet ;
- `ocamlcpt -p options` pour le compilateur natif.

Ces compilateurs produisent le même type de fichiers que les commandes habituelles (7). Les différentes options sont décrites à la figure 10.1.

f	appel de fonctions
i	branches du if
l	boucles while et for
m	branches du match
t	branches du try
a	toutes les options

Figure 10.1 : Options des commandes de profiling

Elles indiquent quelles structures de contrôle doivent être prises en

compte. Par défaut les options fm sont activées.

Exécution d'un programme

Compilateur de code-octet

L'exécution d'un programme compilé dans un mode de profilage produira, si il termine, un fichier nommé ocamlprof.dump qui contient les différentes informations demandées.

On reprend l'exemple sur le produit des entiers d'une liste. On écrit le fichier f1.ml suivant :

```
let rec interval a b =  
  if b < a then []  
  else a::(interval (a+1) b);;  
  
exception Found_zero ;;  
  
let mult_list l =  
  let rec mult_rec l = match l with  
    [] -> 1  
    | 0::_ -> raise Found_zero  
    | n::x -> n * (mult_rec x)  
  in  
  try mult_rec l with Found_zero -> 0  
;;
```

et le fichier f2.ml qui utilise les fonctions de f1.ml :

```
let l1 = F1.interval 1 30;;  
let l2 = F1.interval 31 60;;  
let l3 = l1 @ (0::l2);;  
  
print_int (F1.mult_list l1);;  
print_newline();;  
  
print_int (F1.mult_list l3);;  
print_newline();;
```

La compilation de ces fichiers en mode profilage est la suivante :

```
ocamlcp -i -p a -c f1.ml
val profile_f1_ : int array
val interval : int -> int -> int list
exception Found_zero
val mult_list : int list -> int
```

Avec l'option `-p`, le compilateur ajoute une nouvelle fonction (`profile_f1_`) pour l'initialisation des compteurs du module F1. Il en est de même pour le fichier `f2.ml` :

```
ocamlcp -i -p a -o f2.exe f1.cmo f2.ml
val profile_f2_ : int array
val l1 : int list
val l2 : int list
val l3 : int list
```

Compilateur natif

La compilation en code natif donne le résultat suivant :

```
$ ocamlc -i -p -c f1.ml
val interval : int -> int -> int list
exception Found_zero
val mult_list : int list -> int
$ ocamlc -i -p -o f2nat.exe f1.cmx f2.ml
```

Seule l'option `-p` sans argument est utilisée. L'exécution de `f2nat.exe` produit un fichier de nom `gmon.out`. Celui-ci possède un format traitable par les outils habituels Unix (voir page [??](#)).

Présentation des résultats

Comme les informations recueillies par les deux modes de profiling sont différentes, leur présentation s'en ressent. Dans le premier mode (code-octet), des commentaires sur le nombre de passages dans les structures de contrôle sont ajoutés au texte du programme. Dans le second (natif), on associe à chaque fonction le temps passé

dans son corps et son nombre d'appels.

Compilateur code-octet

La commande `ocamlprof` donne l'analyse des résultats des mesures. Elle utilise les informations contenues dans le fichier `camlprof.dump`. Cette commande prend en entrée le source du programme, lit le fichier de mesure et produit un nouveau texte de programme contenant en commentaires les compteurs demandés.

Sur notre exemple cela donnera :

```
$ ocamlprof fl.ml
```

```
let rec interval a b =
  (* 62 *) if b < a then (* 2 *) []
  else (* 60 *) a::(interval (a+1) b);;

exception Found_zero ;;

let mult_list l =
  (* 2 *) let rec mult_rec l = (* 62 *) match l with
    [] -> (* 1 *) 1
  | 0::_ -> (* 1 *) raise Found_zero
  | n::x -> (* 60 *) n * (mult_rec x)
  in
  try mult_rec l with Found_zero -> (* 1 *) 0
;;
```

Ces compteurs reflètent bien les calculs demandés dans F2. Il y a deux appels à `list_mult` et 62 à la fonction auxiliaire `mult_rec`. L'exploration des branches du filtrage **match** montre les 60 passages dans le cas courant, l'unique passage dans le filtre `[]` au premier calcul, et l'unique passage dans le filtre de tête égal à 0, déclenchant une exception dont le filtrage se voit dans le compteur du **try**.

La commande `camlprof` accepte deux options. La première *-f fichier* indique le nom du fichier contenant les mesures. La seconde *-F chaîne* spécifie une chaîne à ajouter dans les commentaires associés aux structures de contrôle traitées.

Compilateur natif

Pour connaître le temps passé dans les appels des fonctions de multiplication des éléments d'une liste, on écrit le fichier f3.ml suivant :

```
let l1 = F1.interval 1 30;;
let l2 = F1.interval 31 60;;
let l3 = l1 @ (0::l2);;

for i=0 to 100000 do
  F1.mult_list l1;
  F1.mult_list l3
done;;

print_int (F1.mult_list l1);;
print_newline();;

print_int (F1.mult_list l3);;
print_newline();;
```

C'est le même fichier que f2.ml avec une boucle de 100000 itérations.

L'exécution du programme crée le fichier gmon.out. Celui-ci est dans un format lisible par la commande gprof que l'on trouve sur les systèmes Unix. L'appel suivant de gprof affiche les informations de temps passé et le graphe des appels. Comme la sortie est assez longue, on ne montre que la première page qui contient le nom des fonctions appelées au moins une fois et le temps passé dans chacune d'elles.

```
$ gprof f3nat.exe
Flat profile:
```

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	na
92.31	0.36	0.36	200004	1.80	1.80	F1

7.69	0.39	0.03	200004	0.15	1.95	F1
0.00	0.39	0.00	2690	0.00	0.00	ol
0.00	0.39	0.00	302	0.00	0.00	da
0.00	0.39	0.00	188	0.00	0.00	gc
0.00	0.39	0.00	174	0.00	0.00	al
0.00	0.39	0.00	173	0.00	0.00	al
0.00	0.39	0.00	173	0.00	0.00	fl
0.00	0.39	0.00	34	0.00	0.00	ca
0.00	0.39	0.00	30	0.00	0.00	ca
0.00	0.39	0.00	30	0.00	0.00	ga

...

Le premier enseignement est que presque tout le temps d'exécution se déroule dans la fonction `F1_mult_rec_45`, qui correspond à la fonction `F1.mult_rec` du fichier `f1.ml`. On s'aperçoit d'autre part que de nombreuses fonctions sont appelées. Les premières de la liste sont les fonctions de la bibliothèque d'exécution de gestion mémoire (9).

Exercices

Traces d'applications

Cet exercice montre l'évaluation des arguments au moment de l'application.

1. Activer la trace de la fonction `List.fold_left` et évaluer l'expression suivante :

```
List.fold_left (-) 1 [2; 3; 4; 5];;
```

Que vous indique la [trace](#)?

```
# #trace List.fold_left ;;
List.fold_left is now traced.
```

```

# List.fold_left (-) 1 [2; 3; 4; 5] ;;
List.fold_left <-- <fun>
List.fold_left --> <fun>
List.fold_left* <-- <poly>
List.fold_left* --> <fun>
List.fold_left** <-- [<poly>; <poly>; <poly>; <poly>]
List.fold_left <-- <fun>
List.fold_left --> <fun>
List.fold_left* <-- <poly>
List.fold_left* --> <fun>
List.fold_left** <-- [<poly>; <poly>; <poly>]
List.fold_left <-- <fun>
List.fold_left --> <fun>
List.fold_left* <-- <poly>
List.fold_left* --> <fun>
List.fold_left** <-- [<poly>]
List.fold_left <-- <fun>
List.fold_left --> <fun>
List.fold_left* <-- <poly>
List.fold_left* --> <fun>
List.fold_left** <-- []
List.fold_left** --> <poly>
List.fold_left** --> <poly>
List.fold_left** --> <poly>
List.fold_left** --> <poly>
List.fold_left** --> <poly>
- : int = -13

```

2. Définir la fonction `fold_left_int`, identique à `List.fold_left`, mais dont le type est : `(int -> int -> int) -> int -> int list -> int`. [Tracer](#) cette fonction. Pourquoi l'affichage de la trace est-il [différent](#)?

Évaluation des performances

On reprend l'exercice proposé au chapitre 9, page ??, où l'on comparait l'évolution du tas pour deux programmes (l'un récursif terminal et l'autre non) de calcul de nombres premiers. On compare maintenant les temps passés dans chaque fonction en utilisant les outils de profiling et de temps d'exécution. Cet exercice permet d'illustrer l'intérêt de l'expansion en ligne (voir chapitre 7).

1. **Compiler** avec l'option de profiling les deux programmes `erart` et `erant` en utilisant le compilateur de code-octet et le compilateur natif.

```
$ ocamlcp -custom -o era_rt_bc unix.cma interval.ml
$ ocamlcp -custom -o era_nrt unix.cma interval.ml t
$ ocamlcpt -p -o era_rt_nat unix.cmxa interval.ml t
$ ocamlcpt -p -o era_nrt_nat unix.cmxa interval.ml
```

2. **Exécuter** ces programmes en leur passant les nombres 3000 4000 5000 6000 sur la ligne de commande.

```
$ era_rt_bc traceRT-BC 3000 4000 5000 6000
$ era_nrt_bc traceNRT-BC 3000 4000 5000 6000
$ era_rt_nat traceRT-NAT 3000 4000 5000 6000
$ era_nrt_nat traceNRT-NAT 3000 4000 5000 6000
```

3. **Visualiser** les résultats par les commandes `ocamlprof` et `grpof`. Que pouvez-vous **dire** au vu des résultats ?

```
$ ocamlprof eras2.ml
(* fichier eras2.ml *)
```

```
let erart l =
  (* 4 *) let rec erart_aux l r = (* 2436 *) match
    [] -> (* 4 *) List.rev r
  | p::q -> (* 2432 *) erart_aux (List.filter (fun
```

```

in
    erart_aux 1 [] ;;

let erart_go n =
    (* 4 *) erart (Interval.interval (<) (fun x -> (
$ time era_rt.exe a1 3000 4000 5000 6000
0.230u 0.010s 0:00.25 96.0%      0+0k 0+0io 129pf+0w
$ time era_rt_nat a2 3000 4000 5000 6000
0.510u 0.010s 0:00.52 100.0%     0+0k 0+0io 134pf+0w
$ time era_nrt.exe a3 3000 4000 5000 6000
0.220u 0.020s 0:00.24 100.0%     0+0k 0+0io 130pf+0w
$ time era_nrt_nat a4 3000 4000 5000 6000
0.520u 0.010s 0:00.53 100.0%     0+0k 0+0io 134pf+0w
$ visu a1
Nombre de Gc: mineur = 131, majeur = 20
$ visu a2
Nombre de Gc: mineur = 131, majeur = 20
$ visu a3
Nombre de Gc: mineur = 131, majeur = 23
$ visu a4
Nombre de Gc: mineur = 131, majeur = 23

```

Résumé

Ce chapitre a présenté les différents outils d'aide à la réalisation de programmes que fournit la distribution d'Objective CAML.

Le premier outil effectue une analyse statique pour déterminer les dépendances d'un ensemble d'unités de compilation. Ces informations sont ensuite intégrables dans un fichier Makefile permettant de ne compiler que le strict nécessaire suite à une modification dans un ou plusieurs fichiers.

D'autres outils donnent des informations sur l'exécution du programme. En premier lieu, la boucle d'interaction offre une trace de l'exécution ; mais nous avons vu que le polymorphisme imposait des restrictions assez lourdes quant aux valeurs qui sont observables. En fait seules les déclarations globales de valeurs monomorphes sont visibles, ce qui inclut tout de même les

paramètres des fonctions monomorphes et permet de tracer l'exécution des fonctions récursives.

Les derniers outils sont ceux de la tradition des langages de développement sous Unix, à savoir un debugger et un profiler. Le premier permet d'exécuter pas à pas un programme et le second fournit des informations sur ses performances. Tous deux ne sont pleinement utilisables que sous Unix.

Pour en savoir plus

Les résultats produits par la commande `ocamldep` peuvent être visualisés sous forme graphique par l'utilitaire `ocamlviz` que l'on trouve à la page suivante :

Lien: <http://www.cis.upenn.edu/~tjim/ocamlviz/index.html>

`ocamlviz` se sert d'un programme indépendant (`dot`), lui aussi téléchargeable :

Lien: <http://www.research.att.com/sw/tools/graphviz/>

Plusieurs fichiers Makefile génériques pour Objective CAML ont été proposés pour simplifier la tâche de gestion de projets :

Lien: http://caml.inria.fr/FAQ/Makefile_ocaml-eng.html

Lien: http://miss.wu-wien.ac.at/~mottl/ocaml_sources/intro.html

Ceux-ci intègrent la sortie d'`ocamldep`.

On trouve dans [HF96] une évaluation de performance d'une vingtaine d'implantations de langages fonctionnels, dont de nombreux ML, sur un exemple de calcul numérique manipulant de grandes structures de données.

Outils d'analyses lexicale et syntaxique

Introduction

La définition et la mise au point d'outils d'analyse lexicale et syntaxique ont été un important domaine de la recherche en informatique. Ce travail a abouti à la réalisation des générateurs d'analyseurs lexicaux et syntaxiques lex et yacc dont camllex et camlyacc, que nous présentons dans ce chapitre, sont les dignes héritiers. Ces deux outils sont les outils rois du domaine, cependant, d'autres, plus simples, comme les flots ou la bibliothèque d'analyse des expressions rationnelles (ou expressions régulières) Str sont suffisants pour des applications ne nécessitant pas une analyse puissante.

La nécessité de tels outils s'est essentiellement fait sentir dans le domaine de la compilation des langages de programmation évolués, mais d'autres applications peuvent mettre en oeuvre avec profit de tels outils : les systèmes de bases de données qui offrent la possibilité de formuler des requêtes, les tableurs qui définissent le contenu de certaines cases comme résultat de l'évaluation d'une formule. En un mot, toute application dont l'interaction avec l'utilisateur se fait au moyen d'un langage met en oeuvre les analyses lexicale et syntaxique.

À un niveau plus modeste, il est fréquent d'utiliser un format ASCII pour stocker des données : fichiers de configurations systèmes, données d'une feuille de calcul. Là encore, l'exploitation de ces données nécessite une phase d'analyse lexicale et une phase d'analyse syntaxique.

Dans les cas simples comme dans les cas compliqués, le problème que doit résoudre l'analyse lexicale et syntaxique est la transformation d'un flot linéaire de caractères en une donnée à la structure plus riche : une suite de mots, une structure d'enregistrements, l'arbre de syntaxe abstraite d'un programme, etc.

Tout langage possède un vocabulaire (le lexique) et une grammaire permettant d'assembler correctement les éléments du vocabulaire (la syntaxe). Pour qu'une machine, ou un programme, soit capable de traiter correctement un langage, celui-ci doit obéir à des règles lexicales et syntaxiques très précises. Une machine n'a pas « *l'esprit*

de finesse » requis pour prendre une décision face à l'ambiguïté des langues naturelles. Il convient donc de s'adresser à elle selon des règles clairement fixées ne souffrant pas d'exception. Aussi, les notions de lexique et de syntaxe ont-elles reçues des définitions formelles que nous présentons brièvement dans ce chapitre avant d'en montrer les applications.

Plan du chapitre

Ce chapitre présente différents outils de la distribution d'Objective CAML pour l'analyse lexicale et l'analyse syntaxique. Cette dernière suppose en règle générale que la première a été effectuée. Dans la première section, nous présentons un outil simple d'analyse lexicale fourni par le module `Genlex`. Nous entrons ensuite un peu plus dans le détail de la définition des ensembles d'unités lexicales en présentant le formalisme des expressions rationnelles. Nous illustrons leur mise en oeuvre dans le module `Str` et l'outil `ocamllex`. Dans la deuxième section nous définissons les grammaires et détaillons les règles de production de phrase d'un langage pour présenter deux types d'analyse : l'analyse descendante et l'analyse ascendante. Elles sont illustrées par l'utilisation des *Stream* et par l'outil `ocamlyacc`. Les analyses précédentes utilisent des grammaires sans contexte. Nous montrerons comment effectuer une analyse contextuelle avec les *Streams*. Dans la troisième section, nous reprenons l'exemple de l'interprète de BASIC, présenté page ??, en utilisant `ocamllex` et `ocamlyacc` pour implanter les fonctions d'analyse lexicale et syntaxique du langage.

Lexique

L'analyse lexicale est le préalable indispensable à tout traitement des suites de caractères : elle découpe cette suite en une suite de mots dits aussi d'unités lexicales, ou lexèmes.

Le module

Ce module fournit une primitive simple permettant d'analyser une suite de caractères selon plusieurs catégories d'unités lexicales

prédéfinies. Ces catégories sont distinguées par le type :

```
# type token =  
  Kwd of string  
  | Ident of string  
  | Int of int  
  | Float of float  
  | String of string  
  | Char of char ;;
```

On pourra ainsi reconnaître dans une suite de caractères un entier (constructeur `Int`) et récupérer la valeur représentée (argument du constructeur de type `int`). Les chaînes et les caractères reconnus obéissent aux conventions usuelles : une chaîne est délimitée par deux (") et un caractère par deux ('). Un flottant est représenté soit en utilisant la notation pointée (par exemple `0.01`) soit la notation en mantisse et exposant (par exemple `1E-2`). Restent les deux constructeurs `Kwd` et `Ident`. Le constructeur `Ident` désigne la catégorie des identificateurs. Ce sont les noms de variables ou de fonctions des langages de programmation, par exemple. Ils correspondent à n'importe quelle suite de lettres et de chiffres pouvant inclure le caractère souligné (`_`) ou l'apostrophe (`'`). Cette suite ne doit pas commencer par un chiffre. On considère également comme identificateur (pour ce module, du moins) toute suite de symboles d'opérations ou de relations tels `+`, `*`, `>` ou `=`. Enfin, le constructeur `Kwd` définit la catégorie des mots clés qui contient des identificateurs distingués ou des caractères spéciaux.

La seule catégorie paramétrable de cet ensemble d'unités prédéfinies est celle des mots clés. La primitive suivante permet de créer un analyseur lexical en prenant comme ensemble de mots clés la liste passée en premier argument.

```
# Genlex.make_lexer ;;  
- : string list -> char Stream.t -> Genlex.token Stream.
```

On obtient alors une fonction qui prend en entrée un flot de caractères et retourne un flot d'unités lexicales (type `token`).

On peut ainsi obtenir facilement un analyseur lexical pour notre BASIC. On déclare l'ensemble des mots clés :

```
# let keywords =  
  [ "REM"; "GOTO"; "LET"; "PRINT"; "INPUT"; "IF"; "THEN";  
    "-"; "!" ; "+" ; "-" ; "*" ; "/" ; "%";  
    "=" ; "<" ; ">" ; "<=" ; ">=" ; "<>";  
    "&" ; "|" ] ;;
```

On fabrique, avec cet ensemble, la fonction d'analyse lexicale :

```
# let line_lexer l = Genlex.make_lexer keywords (Stream.  
val line_lexer : string -> Genlex.token Stream.t = <fun>  
# line_lexer "LET x = x + y * 3" ;;  
- : Genlex.token Stream.t = <abstr>
```

Le fonction `line_lexer` prend en entrée une chaîne de caractères et produit le flot de lexèmes correspondants.

Utilisation des flots

On peut procéder « à la main » à l'analyse lexicale en manipulant directement des flots.

L'exemple suivant est un analyseur lexical pour les expressions arithmétiques. La fonction `lexer` prend un flot de caractères et construit un flot d'unités lexicales de type *lexeme* `Stream.t`¹. Les espaces, tabulations et sauts de lignes sont éliminés. Pour simplifier, nous ne traitons pas les variables ni les entiers négatifs.

```
# let rec spaces s =  
  match s with parser  
  [<' ' ; rest >] -> spaces rest  
  | [<'\\t' ; rest >] -> spaces rest  
  | [<'\\n' ; rest >] -> spaces rest  
  | [<>] -> ();;  
val spaces : char Stream.t -> unit = <fun>  
# let rec lexer s =  
  spaces s;
```

```

match s with parser
|< '(' >] -> [< 'Lsymbol "(" ; lexer s >]
| [< ')' >] -> [< 'Lsymbol ")" ; lexer s >]
| [< '+' >] -> [< 'Lsymbol "+" ; lexer s >]
| [< '-' >] -> [< 'Lsymbol "-" ; lexer s >]
| [< '*' >] -> [< 'Lsymbol "*" ; lexer s >]
| [< '/' >] -> [< 'Lsymbol "/" ; lexer s >]
| [< '0'..'9' as c;
i,v = lexint (Char.code c - Char.code('0')) >]
-> [<'Lint i ; lexer v>]
and lexint r s =
match s with parser
| [< '0'..'9' as c >]
-> let u = (Char.code c) - (Char.code '0') in lexint (1
| [<>] -> r,s
;;
val lexer : char Stream.t -> lexeme Stream.t = <fun>
val lexint : int -> char Stream.t -> int * char Stream.t

```

La fonction `lexint` est dédiée à l'analyse du morceau de flot de caractères correspondant à une constante entière. Elle est appelée par la fonction `lexer` lorsque celle-ci rencontre un caractère codant un chiffre. La fonction `lexint` consomme alors tous les chiffres consécutifs pour donner la valeur de l'entier correspondant.

Expressions rationnelles

Prenons un peu de hauteur et considérons le problème des unités lexicales d'un point de vue plus théorique.

De ce point de vue, une unité lexicale est un mot. Un mot est formé par la concaténation d'éléments d'un alphabet. Pour ce qui nous concerne, l'alphabet considéré est pris dans l'ensemble des caractères ASCII. Théoriquement, un mot peut ne contenir aucun caractère (le mot vide²) ou ne contenir qu'un seul caractère. L'étude théorique de l'assemblage d'éléments de l'alphabet pour former des éléments du lexique (les lexèmes) a abouti à un formalisme simple connu sous le nom d'expressions rationnelles.

Définition

Une expression rationnelle permet de définir des ensembles de mots. Par exemple : l'ensemble des identificateurs. Son mode de définition est basé sur un certain nombre d'opérations ensemblistes. Soient M et N deux ensembles de mots, on peut former :

1. l'union de M et N que l'on note $M \mid N$.
2. le complément de M que l'on note $\hat{M}$. C'est l'ensemble de tous les mots sauf ceux de M .
3. la concaténation de M et N qui est l'ensemble des mots construits en mettant bout à bout un mot de M et un mot de N . On note simplement MN cet ensemble.
4. on peut itérer l'opération de concaténation sur les mots de l'ensemble M pour obtenir l'ensemble des mots formés d'une suite finie, mais de longueur quelconque, des mots de M . On note M^+ cet ensemble. Il contient tous les mots de M , puis tous les mots formés par la concaténation de deux mots de M , puis de trois mots, etc... Si l'on désire en plus que cet ensemble contienne le mot vide, on note M^* .

Comme cela s'avère utile, on se donne la construction $M?$ qui désigne l'ensemble des mots de M plus le mot vide.

On assimile un caractère au singleton ne contenant que ce seul caractère. L'expression $a \mid b \mid c$ désigne alors l'ensemble contenant les trois mots a , b et c . On utilisera la syntaxe plus compacte : $[abc]$ pour définir cet ensemble. Comme notre alphabet est ordonné (par l'ordre des codes ASCII) on peut également définir un intervalle. Par exemple, l'ensemble des chiffres s'écrit : $[0-9]$. On peut se servir des parenthèses pour grouper des expressions.

Si l'on veut utiliser, pour lui même, un des caractères représentant un opérateur, on le fait précéder du caractère d'échappement $\backslash$. Par exemple $(\backslash)^*$ désigne l'ensemble des suites d'étoiles.

Exemple

Considérons l'alphabet comprenant les chiffres (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) les symboles plus (+) et moins (-), le point (.) et la lettre E.

On peut définir l'ensemble *num* des mots représentant les nombres. Appelons *entiers* l'ensemble défini par $[0-9]^+$. On définit *unum*, l'ensemble des nombres non signés, par :

$\text{entiers}?(.\text{entier})?(E(+|-)?\text{entiers})?$

L'ensemble des nombres signés *num* est alors défini par :

$\text{unum} \mid -\text{unum}$ ou par $-?\text{unum}$

Reconnaissance

Une fois défini un ensemble d'expressions, reste le problème de savoir si une chaîne de caractères ou une de ses sous-chaînes appartient ou non à cet ensemble. Pour cela, il faut traduire la définition formelle de l'ensemble en un programme de reconnaissance et de traitement des expressions. Dans le cas des expressions rationnelles, cette traduction peut être automatisée. De telles techniques de traduction sont mises en oeuvre par le module Genlex, dans la bibliothèque Str et dans l'outil ocamllex que nous présentons dans les deux prochains paragraphes.

La bibliothèque

Ce module contient un type abstrait *regexp* représentant les expressions rationnelles ainsi qu'une fonction *regexp* qui prend une chaîne de caractères décrivant une expression rationnelle, plus ou moins selon la syntaxe décrite ci-dessus, et qui la transforme en sa représentation abstraite.

Ce module contient également un certain nombre de fonctions exploitant les expressions rationnelles et manipulant des chaînes de caractères. La syntaxe des expressions rationnelles pour la bibliothèque Str est donnée figure [11.1](#).

.	n'importe quel caractère sauf <code>\n</code>
*	zéro ou plusieurs

	occurrences de l'expression précédente
+	au moins une occurrence de l'expression précédente
?	zéro ou une occurrence de l'expression précédente
[..]	ensemble de caractères l'intervalle est noté – (exemple [0–9]) le complémentaire est noté ^ (exemple [^A– Z])
^	début de ligne à ne pas confondre avec l'utilisation de ^ comme complémentaire d'un ensemble
\$	fin de ligne
	alternative
(..)	regroupement en une expression composée on pourra alors nommer cette expression composée
<i>i</i>	avec <i>i</i> constante entière, texte filtré par la <i>i</i> -ième expression composée
\	caractère d'échappement on l'utilise lorsque l'on veut filtrer l'un des caractères réservés des expressions rationnelles.

Figure 11.1 : Expressions rationnelles

Exemple

Nous voulons écrire une fonction traduisant des dates au format anglo-saxon en des dates françaises dans un fichier de données. On suppose que le fichier est organisé en lignes de champs et que les composants d'une date anglo-saxonne sont séparés par un point. Nous définissons une fonction qui prend en argument une chaîne (i.e. une ligne du fichier), qui isole la date, la décompose, la traduit et la remplace par cette traduction.

```
# let french_date_of d =
  match d with
  [mm; jj; yy] -> jj^"/"^mm^"/"^yy
  | _ -> failwith "Bad date format" ;;
val french_date_of : string list -> string = <fun>

# let english_date_format = Str.regexp "[0-9]+\.[0-9]+\."
val english_date_format : Str.regexp = <abstr>

# let trans_date l =
  try
    let i=Str.search_forward english_date_format l 0 in
    let d1 = Str.matched_string l in
    let d2 = french_date_of (Str.split (Str.regexp "\.") d1)
    Str.global_replace english_date_format d2 l
  with Not_found -> l ;;
val trans_date : string -> string = <fun>

# trans_date ".....06.13.99....." ;;
- : string = ".....13/06/99....."
```

L'outil

L'outil `ocamllex` est un générateur d'analyseur lexical construit pour Objective CAML sur le modèle de l'outil `lex` dédié au langage C. Il produit un fichier source Objective CAML à partir d'un fichier contenant la description des éléments du lexique à reconnaître sous

forme d'ensembles d'expressions rationnelles. À la description de chacun des éléments lexicaux, le programmeur peut attacher une action de traitement dite action sémantique. Le code engendré manipule un type abstrait *lexbuf* défini dans le module *Lexing*. Ce module contient quelques fonctions de manipulation des buffers lexicaux que le programmeur peut utiliser pour définir ses traitements.

L'usage est de donner aux fichiers de description lexicale l'extension *.mll*. Pour obtenir un source Objective CAML à partir d'un fichier *lex_file.mll* on tape la commande

```
ocamllex lex_file.mll
```

On obtient un fichier *lex_file.ml* contenant le code de l'analyseur correspondant. Ce fichier peut alors être intégré dans une application Objective CAML. À chaque ensemble de règles d'analyse lexicale correspond une fonction prenant en entrée un buffer lexical (type *Lexing.lexbuf*) et retournant la valeur définie par les actions sémantiques. Par conséquent, toutes les actions d'une même règle doivent produire une valeur du même type.

Le format général d'un fichier pour *ocamllex* est

```
{
                                en-tête
}
let ident = regexp
                                ...
rule ruleset1 = parse
                                regexp { action }
                                | ...
                                | regexp { action }
and ruleset2 = parse
                                ...
and ...
{
                                suite-et-fin
}
```

Les deux sections « en-tête » et « suite-et-fin » sont facultatives. Elles

contiennent du code Objective CAML définissant types, fonctions, etc. nécessaires au traitement. La dernière section peut définir des fonctions utilisant les règles d'analyse d'ensemble lexicaux données dans la section médiane. La série de déclarations précédant la définition des règles permet de nommer certaines expressions rationnelles. On utilisera alors leur nom dans la définition des règles.

Exemple

Reprenons notre exemple du BASIC. Nous allons pouvoir y affiner le type des unités lexicales retournées. De surcroît, nous pourrions retrouver la fonction `lexer` définie page ?? avec le même type de sortie (*lexeme*), mais qui prendra en entrée un buffer de type *Lexing.lexbuf*.

```
{
  let string_chars s =
    String.sub s 1 ((String.length s)-2) ;;
}

let op_ar = ['- ' '+ ' '* ' '% ' '/']
let op_bool = ['!' '& ' '|']
let rel = ['=' '<' '>']

rule lexer = parse
  [' '] { lexer lexbuf }

| op_ar { Lsymbol (Lexing.lexeme lexbuf) }
| op_bool { Lsymbol (Lexing.lexeme lexbuf) }

| "<=" { Lsymbol (Lexing.lexeme lexbuf) }
| ">=" { Lsymbol (Lexing.lexeme lexbuf) }
| "<>" { Lsymbol (Lexing.lexeme lexbuf) }
| rel { Lsymbol (Lexing.lexeme lexbuf) }

| "REM" { Lsymbol (Lexing.lexeme lexbuf) }
| "LET" { Lsymbol (Lexing.lexeme lexbuf) }
| "PRINT" { Lsymbol (Lexing.lexeme lexbuf) }
```



```
| "INPUT" { Lsymbol (Lexing.lexeme lexbuf) }
| "IF"    { Lsymbol (Lexing.lexeme lexbuf) }
| "THEN"  { Lsymbol (Lexing.lexeme lexbuf) }

| '-'? ['0'-'9']+ { Lint (int_of_string (Lexing.lexeme lexbuf)) }
| ['A'-'z']+      { Lident (Lexing.lexeme lexbuf) }
| '"' [^ '"' ]* '"' { Lstring (string_chars (Lexing.lexeme lexbuf)) }
```

La traduction de ce fichier par ocamllex fournit la fonction `lexer` de type *Lexing.lexbuf* -> *lexeme*. Nous verrons plus loin comment une telle fonction s'utilise en corrélation avec l'analyse syntaxique (voir page ??).

Syntaxe

Grâce à l'analyse lexicale, nous sommes à même de découper notre flot d'entrée en unités plus structurées : les unités lexicales. Il faut encore savoir assembler ces unités de façon à former des phrases syntaxiquement correctes d'un langage donné. Les règles d'assemblage syntaxique sont définies au moyen de règles de grammaires. C'est un formalisme originaire de la linguistique que les mathématiciens théoriciens des langages et les informaticiens ont su reprendre avec profit. Nous avons déjà vu par anticipation page ??, un exemple de grammaire pour le langage Basic. Nous allons reprendre cet exemple afin d'introduire les concepts de base des grammaires.

Grammaire

Formellement, une grammaire est constituée de quatre éléments :

1. Un ensemble de symboles dits terminaux. Ces symboles représentent les unités lexicales du langage. En Basic, on y trouve les symboles opérateurs et de relations arithmétiques et logiques (+, &, <, <=, ..) ; les mots clés du langage (GOTO, PRINT, IF, THEN, ..) ; les entiers (unité *entier*) et les variables (unité *variable*).
2. Un ensemble de symboles dits non terminaux. Ces symboles

représentent les composantes syntaxique du langage. Par exemple, un programme Basic est composé de lignes, on a donc la composante Ligne ; une ligne peut contenir une Expression, etc.

3. Un ensemble de règles dites de production. Elles décrivent comment s'articulent les symboles terminaux et ceux non terminaux pour produire une composante syntaxique. Une ligne en Basic, est formée de son numéro suivi d'une instruction. C'est le sens de la règle suivante :

Ligne	::=	<i>entier</i> Instruction
Une même composante peut être produite de plusieurs façons.		
On sépare les alternatives par le symbole comme dans		
Instruction	::=	LET <i>variable</i> =
		Expression
		GOTO <i>entier</i>
		PRINT Expression

etc.

4. Enfin, parmi tous les non terminaux, on en distingue un que l'on appelle axiome. C'est sa règle de production qui dénote la règle de production du langage dans son entier.

Production et reconnaissance

Les règles de production peuvent servir à reconnaître qu'une suite de lexèmes appartient à un langage.

Considérons, par exemple un langage simple d'expressions arithmétiques :

Exp	::=	<i>entier</i>	(R1)
		Exp + Exp	(R2)
		Exp * Exp	(R3)
		(Exp)	(R4)

où (R1) (R2) (R3) et (R4) sont les noms que nous donnons à nos règles. Après passage par l'analyse lexicale, l'expression $1 * (2+3)$ devient la suite de lexèmes :

$entier * (entier + entier)$

Pour analyser cette phrase et reconnaître qu'elle appartient bien au langage des expressions arithmétiques, nous allons utiliser nos règles de droite à gauche : si une sous-expression correspond à un membre droit de l'une des règles, nous la remplaçons par le membre gauche correspondant et nous recommençons jusqu'à avoir réduit notre expression au non terminal axiome (ici, Exp). Voici les étapes d'une telle analyse³.

$entier * (entier +$	$\neg^{3/4}(R1)$	$Exp * (entier + entier$
$entier)$		$)$
	$\neg^{3/4}(R1)$	$Exp * (Exp + entier)$
	$\neg^{3/4}(R1)$	$Exp * (Exp + Exp)$
	$\neg^{3/4}(R2)$	$Exp * (Exp)$
	$\neg^{3/4}(R4)$	$Exp * Exp$
	$\neg^{3/4}(R3)$	Exp

En partant de la dernière ligne qui ne contient plus que Exp et en remontant les flèches, on lit comment a pu être produite notre expression en partant de la règle axiome Exp : c'est donc une phrase bien formée du langage défini par la grammaire.

La traduction des grammaires en programmes capables de reconnaître qu'une suite de lexèmes appartient au langage défini par une grammaire est un problème beaucoup plus complexe que celui de l'utilisation des expressions rationnelles. En effet, un résultat mathématique nous dit que tout ensemble (de mots) défini au moyen du formalisme des expressions rationnelles peut aussi être défini dans un autre formalisme : les automates finis déterministes. Et ces derniers sont faciles à exprimer comme des programmes prenant en entrée une suite de caractères. On ne dispose pas d'un tel résultat pour le cas des grammaires en général. Cependant, on dispose de résultats plus faibles établissant l'équivalence entre certaines classes de grammaires et des automates un peu plus riches : les automates à pile. Nous ne voulons pas entrer dans les détails de ces résultats ni donner une définition exacte de ce que sont les automates. Cependant, nous ne pourrions pas faire l'économie d'une petite étude des grammaires afin d'identifier quelle classe de grammaire peut être utilisée dans les outils de génération d'analyseurs syntaxiques ou pour implanter directement un

analyseur.

Analyse descendante

La décomposition de l'expression $1 * (2+3)$ proposée au paragraphe précédent n'est pas unique : on aurait pu tout aussi bien commencer par réduire les *entiers* de droite à gauche, ce qui aurait permis d'utiliser plutôt la règle (R2) pour réduire d'abord $2+3$. Ces deux façons de faire constituent deux types d'analyses : l'analyse ascendante (de droite à gauche) et l'analyse descendante (de gauche à droite). Cette dernière est facilement réalisable à partir de flots de lexèmes en utilisant le module Stream. L'analyse ascendante est celle réalisée par l'outil *ocamlyacc*, elle met en jeu un mécanisme explicite de pile comme cela a déjà été illustré pour l'analyse syntaxique des programmes Basic. Le choix du type d'analyse n'est pas indifférent car suivant la forme de grammaire utilisée pour spécifier un langage, on peut ou non utiliser l'analyse descendante.

Le cas simple

Le cas d'école de l'analyse descendante est la notation préfixée des expressions arithmétiques définie par :

Expr	::=	<i>entier</i>
		+ Expr Expr
		* Expr Expr

Dans un tel cas, il suffit de connaître le premier lexème pour savoir quelle règle de production a pu être utilisée. Cette prédictibilité immédiate permet de ne pas gérer explicitement la pile de l'analyse et de s'en remettre à l'empilement des appels récursifs de l'analyseur. Il est alors très facile d'écrire un programme implantant l'analyse descendante en utilisant les fonctionnalités des modules Genlex et Stream. La fonction *infix_of* ainsi obtenue prend une expression préfixée et retourne son équivalent infixé.

```
# let lexer s =  
  let ll = Genlex.make_lexer ["+"; "*"]  
  in ll (Stream.of_string s) ;;
```

```

val lexer : string -> Genlex.token Stream.t = <fun>
# let rec stream_parse s =
  match s with parser
  [<'Genlex.Ident x>] -> x
  | [<'Genlex.Int n>] -> string_of_int n
  | [<'Genlex.Kwd "+"; e1=stream_parse; e2=stream_parse>]
  | [<'Genlex.Kwd "*"; e1=stream_parse; e2=stream_parse>]
  | [<>] -> failwith "Parse error"
;;
val stream_parse : Genlex.token Stream.t -> string = <fu
# let infix_of s = stream_parse (lexer s) ;;
val infix_of : string -> string = <fun>
# infix_of "* +3 11 22";;
- : string = "((3+11)*22)"

```

Il faut prendre garde à ce que l'analyse lexicale est assez fruste. Il est conseillé d'intercaler systématiquement un espace entre les diverses unités lexicales de la chaîne d'entrée.

```

# infix_of "*+3 11 22";;
- : string = "*+"

```

Un cas moins simple

L'analyse syntaxique utilisant les flots est prédictive. Elle impose deux conditions sur les grammaires.

1. Il ne doit pas y avoir de récursivité gauche dans les règles de grammaire. Une règle est récursive à gauche lorsque que le membre droit commence par le non-terminal qui est le membre gauche de la règle, comme dans $\text{Exp} ::= \text{Exp} + \text{Exp}$;
2. Il ne doit pas y avoir deux règles commençant par la même expression.

La grammaire usuelle des expressions arithmétiques donnée page ?? ne convient pas directement à l'analyse descendante : elle ne satisfait aucun des deux critères ci-dessus. Pour pouvoir utiliser une analyse descendante, il faut reformuler la grammaire de façon à supprimer la récursion gauche et le non déterminisme des règles.

Pour les expressions arithmétiques, on peut donner, par exemple :

Expr	::=	Atom SuiteExpr
SuiteExpr	::=	+ Atom
		- Atom
		* Atom
		/ Atom
		e
Atom	::=	<i>entier</i>
		(Expr)

Remarquons l'usage du mot vide e dans la définition de SuiteExp. Il est nécessaire si nous voulons qu'un entier tout seul soit une expression.

Notre grammaire permet l'implantation de l'analyseur suivant qui n'est qu'une traduction simple des règles de production. Cet analyseur produit l'arbre de syntaxe abstraite des expressions arithmétiques.

```
# let rec suite = parser
  [< 'Lsymbol "+"; e2 = atom >] -> Some (PLUS,e2)
  | [< 'Lsymbol "-"; e2 = atom >] -> Some (MOINS,e2)
  | [< 'Lsymbol "*"; e2 = atom >] -> Some (MULT,e2)
  | [< 'Lsymbol "/"; e2 = atom >] -> Some (DIV,e2)
  | [< >] -> None
and atom = parser
  [< 'Lint i >] -> ExpInt i
  | [< 'Lsymbol "("; e = expr ; 'Lsymbol ")" >] -> e
and expr s =
  match s with parser
  [< e1 = atom >] ->
  match suite s with
  None -> e1
  | Some (op,e2) -> ExpBin(e1,op,e2) ;;
val suite : lexeme Stream.t -> (op_bin * expression) option
val atom : lexeme Stream.t -> expression = <fun>
val expr : lexeme Stream.t -> expression = <fun>
```

La difficulté d'utilisation de l'analyse descendante vient de la nécessité d'avoir des grammaires d'une forme très restreinte. Et

lorsque le langage visé se formule naturellement, comme dans le cas des expressions infixées, avec une grammaire récursive à gauche, il n'est pas toujours trivial de trouver une grammaire équivalente (i.e. qui définit le même langage) qui satisfasse aux exigences de l'analyse descendante. C'est pourquoi les outils yacc et ocamlyacc mettent plutôt en oeuvre un mécanisme d'analyse ascendante qui autorise la définition de grammaires plus naturelles. Nous allons cependant voir que là encore, tout ne sera pas possible.

Analyse ascendante

Page ??, nous avons présenté intuitivement les principes de l'analyse ascendante : avancer (shift) ou réduire (reduce). À chacune de ces actions l'état de la pile est modifié. On peut déduire cette suite d'actions des règles de grammaire, si toutefois, comme dans le cas de l'analyse descendante, la grammaire s'y prête ! Ici encore, la difficulté viendra du non déterminisme des règles qui ne permettra pas de choisir entre avancer et réduire. Nous allons illustrer le fonctionnement de l'analyse ascendante et ses causes d'échec en considérant les sempiternelles expressions arithmétiques en notation postfixée et en notation infixée.

Pour le meilleur

La grammaire simplifiée des expressions arithmétiques postfixées est :

Expr	::=	<i>chiffre</i>	(R1)
		Expr Expr +	(R2)
		Expr Expr *	(R3)

Cette grammaire est duale de celle des expressions préfixées : il faut attendre la fin de l'analyse pour savoir, en fin de compte, quelle règle a été utilisée, mais une fois cette fin atteinte, on sait exactement comment faire. En fait, l'analyse ascendante de telles expressions est extrêmement proche d'un mécanisme d'évaluation avec une pile. Simplement, au lieu d'empiler les valeurs résultant du calcul, on empile les symboles de la grammaire. L'idée étant que si

l'on démarre avec une pile vide il faut obtenir une pile ne contenant que le non terminal axiome lorsque l'entrée est épuisée. Les modifications de la pile sont les suivantes : si l'on avance, on empile le non terminal courant ; si l'on réduit, c'est que les premiers éléments de la pile correspondent à un membre droit (inversé) de règle, auquel cas on remplace ces éléments par le membre gauche correspondant.

La figure 11.2 illustre comment l'analyse ascendante traite l'expression : $1\ 2\ +\ 3\ *4\ +$. L'unité lexicale lue est soulignée. La fin de l'entrée est marquée par un \$.

Action	Entrée	Pile
	<u>1 2 + 3 * 4 +</u> \$	[]
Avancer	2 + 3 * 4 + \$	[1]
Réduire (R1)	2 + 3 * 4 + \$	[Expr]
Avancer	+ 3 * 4 + \$	[2Expr]
Réduire (R1)	+ 3 * 4 + \$	[Expr Expr]
Avancer, Réduire (R2)	3 * 4 + \$	[Expr]
Avancer, Réduire (R1)	* 4 + \$	[Expr Expr]
Avancer, Réduire (R3)	4 + \$	[Expr]
Avancer, Réduire (R1)	+ \$	[Expr Expr]
Avancer, Réduire (R2)	\$	[Expr]

Figure 11.2 : Exemple d'analyse ascendante

Pour le pire

Toute la difficulté du passage de la grammaire au programme de reconnaissance du langage est de déterminer quel type d'action appliquer. Nous allons illustrer cette difficulté sur trois exemples provoquant trois genres d'indétermination.

Le premier exemple est une grammaire pour les expressions n'utilisant que l'addition :

$E0$	$:: =$	<i>entier</i>	(R1)
		$E0 + E0$	(R2)

L'indécision de cette grammaire vient de la règle (R2). Supposons la situation suivante:

Action	Entrée	Pile
:		
	+ ...	[E0 + E0 ...]
:		

Dans un tel cas, il n'est pas possible de déterminer s'il faut avancer et empiler le + ou réduire selon (R2) les deux E0 et le + présents dans la pile. Nous sommes en présence d'un conflit avancer-réduire (shift/reduce). Il provient du fait que l'expression *entier + entier + entier* peut être produite, par dérivation droite, de deux façons.

Première façon : E0	$\frac{3}{4} \textcircled{R}(R2)$	$E0 + E0$
	$\frac{3}{4} \textcircled{R}(R1)$	$E0 + \textit{entier}$
	$\frac{3}{4} \textcircled{R}(R2)$	$E0 + E0 +$
		<i>entier</i>
etc.		
Seconde façon : E0	$\frac{3}{4} \textcircled{R}(R2)$	$E0 + E0$
	$\frac{3}{4} \textcircled{R}(R2)$	$E0 + E0 + E0$
	$\frac{3}{4} \textcircled{R}(R1)$	$E0 + E0 +$
		<i>entier</i>
etc.		

Les expressions obtenues par les deux dérivations peuvent sembler similaire du point de vue de l'évaluation d'une expression :

$(entier + entier) + entier$ et $entier + (entier + entier)$

mais différent pour la construction d'un arbre syntaxique (voir figure 6.3 page ??).

Le deuxième exemple de grammaire engendrant un conflit entre avancer et réduire relève du même genre d'ambiguïté : un parenthésage implicite. Mais contrairement au cas précédent, le choix en avancer et réduire modifie le sens de l'expression analysée. Soit donc la grammaire :

E1	::=	<i>entier</i>	(R1)
		$E1 + E1$	(R2)
		$E1 * E1$	(R3)

On retrouve avec cette grammaire le même conflit que ci-dessus à la fois pour + et *. Mais s'y rajoute un conflit entre + et *. Ici encore, une même expression peut être produite de deux façons. Il existe deux dérivations droite de

*entier + entier * entier*

Première façon : E1	$\frac{3}{4} \textcircled{R}(R3)$	$E1 * E1$
	$\frac{3}{4} \textcircled{R}(R1)$	$E1 * \textit{entier}$
	$\frac{3}{4} \textcircled{R}(R2)$	$E1 + E1 * \textit{entier}$
etc.		
Seconde façon : E1	$\frac{3}{4} \textcircled{R}(R2)$	$E1 + E1$
	$\frac{3}{4} \textcircled{R}(R3)$	$E1 + E1 * E1$
	$\frac{3}{4} \textcircled{R}(R1)$	$E1 + E1 * \textit{entier}$
etc.		

Ici, les deux paranthésages (implicites) ne sont pas équivalents :

$(entier + entier) * entier$ ¹ $entier + (entier * entier)$

Ce problème a déjà été rencontré pour les expressions Basic (voir page ??). Il a été résolu en attribuant des priorités aux opérateurs : on réduit (R3) avant (R2). Ce qui revient à parenthéser les produits.

On peut également résoudre le problème du choix entre + et * en

modifiant la grammaire. On introduit deux nouveaux terminaux : T, pour terme, et F, pour facteur. Ce qui donne :

E	::=	E + T	(R1)
		T	(R2)
T	::=	T * F	(R3)
		F	(R4)
F	::=	<i>entier</i>	(R5)

Il n'y a maintenant plus qu'une seule façon de démarrer la séquence de production de *entier + entier*entier* : en utilisant la règle (R1).

Le troisième et dernier exemple concerne les instructions conditionnelles des langages de programmation. Un langage comme Pascal offre deux conditionnelles : le if .. then et le if .. then .. else. Imaginons la grammaire suivante :

Instr	::=	if Exp then Instr (R1)
		if Exp then Instr (R2)
		else Instr
		etc...

Et dans la situation suivante :

Action	Entrée	Pile
:		
	else...	[Instr then Exp if...]
:		

On ne peut pas déterminer si les premiers éléments de la pile correspondent à la conditionnelle de (R1), au quel cas il faut réduire, ou au premier Instr de la règle (R2), auquel cas il faut avancer.

Outre les conflits avancer-réduire, l'analyse ascendante provoque également des conflits réduire-réduire.

Nous présentons maintenant l'outil ocamlyacc qui utilise cette technique et peut rencontrer ces conflits.

L'outil

L'outil `ocamlyacc` est construit sur le même principe que `ocamllex` : il prend en entrée un fichier de description d'une grammaire dont chaque règle est attachée à une action sémantique et il engendre deux fichiers Objective CAML d'extension `.ml` et `.mli` contenant la fonction d'analyse syntaxique et son interface.

Format général

Les fichiers de description syntaxique pour `ocamlyacc` utilisent par convention l'extension `.mly` et ont le format suivant :

```
%{
                                en-tête
}%
                                déclarations
%%
                                règles
%%
                                suite-et-fin
```

Le format d'une règle est :

```
non-terminal      :      symbole...symbole action
                    |      sémantique }
                    |
                    |      ...
                    |      symbole...symbole action
                    |      sémantique }
                    ;
```

Un symbole est soit un terminal, soit un non terminal. Les sections « en-tête » et « suite-et-fin » jouent le même rôle qu'en `ocamllex` à la différence près que l'en-tête n'est visible que des règles et non des déclarations. En particulier, cela entraîne que les ouvertures de module (**open**) ne sont pas prises en compte dans la partie déclarative et donc que les types doivent être qualifiés.

Actions sémantiques

Les actions sémantiques sont des morceaux de code Objective CAML exécutés lorsque l'analyseur réduit la règle à laquelle elles sont associées. Le corps d'une action sémantique peut faire référence aux composants du membre droit de la règle. Ceux-ci sont numérotés de gauche à droite en commençant par 1. Le premier composant est référencé par \$1, le deuxième par \$2, etc.

Axiomes

On peut déclarer plusieurs non terminaux axiomes de la grammaire en écrivant, dans la partie déclaration :

```
%start non-terminal .. non-terminal
```

Pour chacun d'eux sera engendré une fonction d'analyse. Il faut préciser, toujours dans la partie déclaration, quel sera le type de retour de ces fonctions.

```
%type <type-retour> non-terminal
```

Le `type-retour` doit être qualifié.

Warning

Les symboles non terminaux deviennent les noms des fonctions d'analyse. Ils ne doivent donc pas commencer par une majuscule qui est l'initiale réservée aux constructeurs.

Unités lexicales

Les règles de grammaire font référence aux unités lexicales qui sont les symboles terminaux des règles.

Un (ou des) lexème(s) se déclare(nt) de la façon suivante :

```
%token PLUS MOINS MULT DIV MOD
```

Certaines unités lexicales, comme les identificateurs, représentent

un ensemble de chaînes de caractères. Lorsque l'on rencontre un identificateur on veut pouvoir récupérer cette chaîne. On indique à l'analyseur lexical que ces lexèmes ont une valeur associée en donnant entre `<` et `>` le type de cette valeur :

```
%token <string> IDENT
```

Warning

Après traitement par `ocamlyacc` toutes ces déclarations sont transformées en constructeurs du type *token*. Ils doivent donc impérativement commencer par une majuscule.

On peut utiliser des chaînes de caractères comme terminaux implicites comme dans :

```
expr : expr "+" expr    { ... }  
      | expr "*" expr    { ... }  
      | ...  
      ;
```

auquel cas, il est inutile de déclarer un symbole les représentant : ils sont directement traités par l'analyseur syntaxique sans avoir à passer par l'analyseur lexical. Par soucis d'uniformité, nous ne conseillons cependant pas cette manière de faire.

Priorité, associativité

Nous avons vu que nombre de conflits de l'analyse ascendante proviennent de règles implicites d'association d'un opérateur ou de conflits de priorité entre opérateurs. Pour éliminer ces conflits, on peut déclarer des règles d'associativité par défaut (à gauche ou à droite ou aucune) pour les opérateurs ainsi que des règles de priorité. La déclaration suivante indique que les opérateurs `+` (lexème `PLUS`) et `*` (lexème `MULT`) sont associatifs à droite par défaut et que `*` a une priorité supérieure à celle de `+` car `MULT` est déclaré après `PLUS`.

```
%left PLUS
```

%left MULT

Deux opérateurs déclarés sur une même ligne ont même priorité.

Options de la commande

ocamlyacc possède deux options :

- -b nom : les fichiers Objective CAML engendrés sont nom.ml et nom.mli;
- -v : crée un fichier d'extension .output contenant la numérotation des règles, les états de l'automate reconnaissant la grammaire et les sources de conflits.

Utilisation conjointe avec ocamllex

On peut composer les deux outils ocamllex et ocamlyacc de sorte que la transformation du flot de caractères en flot de lexèmes soit l'entrée de l'analyseur syntaxique. Pour se faire, Le type *lexeme* doit être commun aux deux. Ce type est défini dans les fichiers d'extension .mli et .ml engendrés par ocamlyacc à partir des déclarations des **token** du fichier d'extension .mly correspondant. Le fichier .mll importe ce type; ocamllex traduit ce fichier en une fonction Objective CAML de type *Lexing.lexbuf -> lexeme*. L'exemple page ?? illustre cette interaction et décrit les différentes phases de compilation.

Grammaires contextuelles

Les analyseurs engendrés par ocamlyacc traitent de langages engendrés par des grammaires dites non contextuelles. La poursuite de l'analyse d'un flot de lexème ne dépend pas des valeurs syntaxiques déjà traitées. Ce n'est pas le cas du langage *L* décrit par la formule suivante :

$$L ::= wCw \mid w \text{ avec } w \in (A|B)^*$$

où A , B et C sont des symboles terminaux. On a écrit wCw (avec $w \hat{=} (A|B)^*$) et non simplement $(A|B)^*C(A|B)^*$ car nous voulons avoir le *même* mot à gauche et à droite du C médian.

Pour analyser les mots de L , il faut garder en mémoire ce qui a été rencontré avant la lettre C pour vérifier que l'on retrouve bien la même chose après. Voici une solution à ce problème qui utilise le parcours d'un flot. L'idée générale de l'algorithme est de construire une fonction d'analyse de flot qui reconnaîtra exactement le sous-mot figurant avant l'éventuelle occurrence de C :

On se donne le type :

```
# type token = A | B | C ;;
```

La fonction `parse_w1` construit la fonction de mémorisation du premier w sous forme d'une liste de fonctions d'analyse de flot atomique (i.e. pour un seul *token*) :

```
# let rec parse_w1 s =  
  match s with parser  
    [<'A; l = parse_w1 >] -> (parser [<'A >] -> "a")::l  
    | [<'B; l = parse_w1 >] -> (parser [<'B >] -> "b")::l  
    | [< >] -> [] ;;  
val parse_w1 : token Stream.t -> (token Stream.t -> string) list
```

Le résultat des fonctions construites par `parse_w1` est simplement la chaîne de caractères contenant l'unité lexicale analysée.

La fonction `parse_w2` prend en argument une liste construite par `parse_w1` pour composer chacun de ses éléments en une seule fonction d'analyse :

```
# let rec parse_w2 l =  
  match l with  
    p::pl -> (parser [< x = p; l = (parse_w2 pl) >] -> x^l)  
    | [] -> parser [<>] -> "" ;;  
val parse_w2 : ('a Stream.t -> string) list -> 'a Stream.t -> string
```

Le résultat de l'application de `parse_w2` sera la chaîne représentant le sous-mot w . Par construction, la fonction `parse_w2` ne pourra

reconnaître que le sous-mot parcouru par `parse_w1`.

En utilisant la possibilité de nommer dans les flots les résultats intermédiaires, on écrit la fonction de reconnaissance des mots du langage *L* :

```
# let parse_L = parser [< l = parse_w1 ; 'C; r = (parse_
val parse_L : token Stream.t -> string = <fun>
```

Voici deux petits exemples. Le premier donne en résultat la chaîne contenant le mot entourant *C*, le second échoue car les deux mots entourant *C* sont différents :

```
# parse_L [< 'A; 'B; 'B; 'C; 'A; 'B; 'B >];;
- : string = "abb"
# parse_L [< 'A; 'B; 'C; 'B; 'A >];;
Uncaught exception: Stream.Error("")
```

Basic revisité

Nous voulons à présent utiliser le couple `ocamllex` et `ocamlyacc` afin de remplacer la fonction `parse` donnée page ?? pour `Basic`, par des fonctions engendrées à partir des fichiers de spécification du lexique et de la syntaxe du langage.

Pour y arriver, nous ne pourrons pas reprendre tels quels le type des unités lexicales que nous avons défini. Nous allons devoir définir un type plus précis permettant de distinguer les différents opérateurs, les commandes et les mots clés.

Nous aurons également besoin d'isoler les déclarations de type concernant la syntaxe abstraite dans un fichier `basic_types.mli`. Il contiendra les déclarations du type phrases et de tous les types nécessaires à celui-là.

Le fichier

L'en-tête

du fichier contient un appel aux déclarations de types pour la syntaxe abstraite ainsi que deux fonctions auxiliaires de conversion de chaînes de caractères vers leur équivalent en syntaxe abstraite.

```
%{  
open Basic_types ;;  
  
let phrase_of_cmd c =  
  match c with  
    "RUN" -> Run  
  | "LIST" -> List  
  | "END" -> End  
  | _ -> failwith "line : unexpected command"  
;;  
  
let op_bin_of_rel r =  
  match r with  
    "=" -> EGAL  
  | "<" -> INF  
  | "<=" -> INFEQ  
  | ">" -> SUP  
  | ">=" -> SUPEQ  
  | "<>" -> DIFF  
  | _ -> failwith "line : unexpected relation symbol"  
;;  
  
%}
```

Les déclarations

contiennent trois parties : la déclaration des lexèmes; la déclaration de leurs règles d'associativité et de priorité; la déclaration de la règle axiome line qui correspond à l'analyse d'une ligne de programme ou de commande.

Les unités lexicales sont les suivantes :

```
%token <int> Lint
%token <string> Lident
%token <string> Lstring
%token <string> Lcmd
%token Lplus Lmoins Lmult Ldiv Lmod
%token <string> Lrel
%token Land Lor Lneg
%token Lpar Rpar
%token <string> Lrem
%token Lrem Llet Lprint Linput Lif Lthen Lgoto
%token Legal
%token Leol
```

Leurs noms parlent d'eux-mêmes et ils sont décrits par le fichier `basic_lexer.mll` (voir page ??).

Les règles de priorité entre opérateurs reprennent les valeurs données par les fonctions `priority_ob` et `priority_ou` définies lorsque nous avons donnée la grammaire de notre Basic (voir page ??).

```
%right Lneg
%left Land Lor
%left Legal Lrel
%left Lmod
%left Lplus Lmoins
%left Lmult Ldiv
%nonassoc Lopp
```

Le symbole `Lopp` nous servira à traiter le moins unaire. Ce n'est pas un terminal de la grammaire, mais un « pseudo non terminal » permettant de gérer la surcharge entre opérateurs lorsque deux utilisations d'un même symbole ne doivent pas avoir la même priorité selon le contexte. Et c'est le cas du symbole moins (-). Nous reviendrons sur ce point lorsque nous aurons énoncé les règles de la grammaire.

Le non terminal axiome est `line`. La fonction engendrée retournera l'arbre de syntaxe abstraite de la ligne analysée.

```
%start line
%type <Basic_types.phrase> line
```

Les règles

de grammaire se décomposent en trois non terminaux : line pour une ligne; inst pour une instruction du langage; exp pour les expressions. L'action associée à chacune des règles se contente de construire le morceau de syntaxe abstraite correspondant.

```
%%
line :
  Lint inst Leol                                { Ligne {num=$1; inst=$2}
  | Lcmd          Leol                          { phrase_of_cmd $
  ;

inst :
  Lrem                                { Rem $1 }
  | Lgoto Lint                       { Goto $2 }
  | Lprint exp                       { Print $2 }
  | Linput Lident                    { Input $2 }
  | Lif exp Lthen Lint               { If ($2, $4) }
  | Llet Lident Legal exp            { Let ($2, $4) }
  ;

exp :
  Lint                                { ExpInt $1 }
  | Lident                           { ExpVar $1 }
  | Lstring                          { ExpStr $1 }
  | Lneg exp                         { ExpUnr (NON, $2) }
  | exp Lplus exp                    { ExpBin ($1, PLUS, $3) }
  | exp Lmoins exp                   { ExpBin ($1, MOINS, $3) }
  | exp Lmult exp                    { ExpBin ($1, MULT, $3) }
  | exp Ldiv exp                     { ExpBin ($1, DIV, $3) }
  | exp Lmod exp                     { ExpBin ($1, MOD, $3) }
  | exp Legal exp                    { ExpBin ($1, EGAL, $3) }
  | exp Lrel exp                     { ExpBin ($1, (op, $3) }
  | exp Land exp                     { ExpBin ($1, ET, $3) }
  | exp Lor exp                      { ExpBin ($1, OU, $3) }
```

```

| Lmoins exp %prec Lopp          { ExpUnr(OPPOSE, $2) }
| Lpar exp Rpar                  { $2 }
;
%%

```

Ces règles n'appellent pas de commentaire particulier sauf celle-ci :

exp :

```

...
| Lmoins exp %prec Lopp { ExpUnr(OPPOSE, $2) }

```

Elle concerne l'utilisation unaire du symbole -. Le mot clé %prec que l'on trouve dans cette règle indique que cette construction doit recevoir la priorité de Lopp (ici, la priorité maximale).

Le fichier

L'analyse lexicale ne contient qu'un seul ensemble : lexer qui correspondra au type près à l'ancienne fonction lexer (voir page ??). L'action sémantique associée à la reconnaissance de chaque unité lexicale est simplement le renvoi du constructeur correspondant. Comme le type des unités lexicales est déclaré dans le fichier des règles de syntaxe, il faut inclure ce dernier. On rajoute une petite fonction auxiliaire chargée de supprimer les guillemets autour des chaînes de caractères.

```

{
  open Basic_parser ;;

  let string_chars s = String.sub s 1 ((String.length s)-
}

rule lexer = parse
  [' ' '\t']          { lexer lexbuf }

  | '\n'              { Leol }

  | '!'               { Lneg }
  | '&'               { Land }

```

' '	{ Lor }
'='	{ Legal }
'%'	{ Lmod }
'+'	{ Lplus }
'-'	{ Lmoins }
'*'	{ Lmult }
'/'	{ Ldiv }
['<' '>']	{ Lrel (Lexing.lexeme lexbuf) }
"<="	{ Lrel (Lexing.lexeme lexbuf) }
">="	{ Lrel (Lexing.lexeme lexbuf) }
"REM" [^ '\n']*	{ Lrem (Lexing.lexeme lexbuf) }
"LET"	{ Llet }
"PRINT"	{ Lprint }
"INPUT"	{ Linput }
"IF"	{ Lif }
"THEN"	{ Lthen }
"GOTO"	{ Lgoto }
"RUN"	{ Lcmd (Lexing.lexeme lexbuf) }
"LIST"	{ Lcmd (Lexing.lexeme lexbuf) }
"END"	{ Lcmd (Lexing.lexeme lexbuf) }
['0'-'9']+	{ Lint (int_of_string (Lexing.lexeme lexbuf)) }
['A'-'z']+	{ Lident (Lexing.lexeme lexbuf) }
''' [^ ''']* '''	{ Lstring (string_chars (Lexing.lexeme lexbuf)) }

Remarquons que nous avons dû isoler le symbole = qui sert à la fois dans les expressions et dans l'instruction d'affectation.

Seules deux de ces expressions rationnelles nécessitent des explications complémentaires. La première concerne les lignes de commentaires ("REM" [^ '\n']*). Elle reconnaît le mot clé REM suivi par un nombre quelconque de caractères différents du retour chariot. La seconde concerne les chaîne de caractères (''' [^ ''']* ''') considérées comme une suite de caractères différents de " et encadrée par des ".

Compilation, intégration

La compilation du couple analyseur lexical, analyseur syntaxique doit être faite en suivant un certain ordre. Ceci est dû à la dépendance mutuelle dans la déclaration des lexèmes. Ainsi, dans notre exemple, il faudra taper la séquence de commandes :

```
ocamlc -c basic_types.mli
ocamlyacc basic_parser.mly
ocamllex basic_lexer.mll
ocamlc -c basic_parser.mli
ocamlc -c basic_lexer.ml
ocamlc -c basic_parser.ml
```

Ce qui engendrera les fichiers `basic_lexer.cmo` et `basic_parser.cmo` qui pourrons être intégrés à l'application.

Nous disposons maintenant de tout le matériel nécessaire pour reformuler notre application.

Nous supprimons tous les types et toutes les fonctions des paragraphes « analyse lexicale » (page ??) et << analyse syntaxique >> (page ??) de l'application Basic et, dans la fonction `une_commande` (page ??), nous remplaçons l'expression

```
match parse (input_line stdin) with
```

par

```
match line lexer (Lexing.from_string ((input_line stdin
```

Remarquons que nous devons remettre en fin de ligne le caractère `'\n'` que la fonction `input_line` avait supprimé. Ceci est nécessaire, car nous utilisons ce caractère pour marquer la fin d'une ligne de commande (Leol).

Exercices

Suppression de commentaires

Les commentaires en Objective CAML sont hiérarchiques. On peut ainsi commenter des parties de texte, y compris celles contenant déjà des commentaires. Un commentaire commence par les caractères `(*` et se termine par `*)`. Voici un exemple :

```
(* début du commentaire
sur plusieurs
lignes *)

let succ x = (* fonction successeur *)
x + 1;;

(* niveau 1  texte commenté
let old_succ y = (* niveau 2 fonction successeur niveau
y +1;;
niveau 1 *)
succ 2;;
```

Le but de cet exercice est de créer un nouveau texte sans tous les commentaires. Le choix de l'outil d'analyse lexicale est libre.

1. Écrire un [analyseur lexical](#) qui puisse reconnaître les commentaires d'Objective CAML. Ceux-ci commencent par un `(*` et se terminent par un `*)`. Attention les commentaires sont bien balancés, c'est-à-dire que le nombre d'ouvertures est égal au nombre de fermetures.

```
(** fichier comment1.mll **)

{
let ignore_lex lb = ignore (Lexing.lexeme lb)
let traite_normal = ignore_lex
let traite_comment = ignore_lex
exception Commentaires_mal_balances
```



```

}

rule normal = parse
  "(*"      { ignore_lex lexbuf ; commentaire lexbuf ;
|  "*)"     { raise Commentaires_mal_balances }
|  _        { traite_normal lexbuf ; normal lexbuf }
|  eof      { () }

```

```

and commentaire = parse
  "(*"      { ignore_lex lexbuf ; commentaire lexbuf ;
|  "*)"     { ignore_lex lexbuf }
|  _        { traite_comment lexbuf ; commentaire le
|  eof      { raise Commentaires_mal_balances }

```

2. Écrire un [programme](#) qui prend un fichier, le lit, retire les commentaires de son texte puis crée un nouveau fichier résultat.

```

(** fichier comment2.mll **)
{
  let ignore_lex lb = ignore (Lexing.lexeme lb)
  let sortie = ref stdout
  let init_sortie f = sortie := f
  let traite_normal lb = output_string !sortie (Lexi
  let traite_comment = ignore_lex
  exception Commentaires_mal_balances
}

```

```

rule normal = parse
  "(*"      { ignore_lex lexbuf ; commentaire lexbuf ;
|  "*)"     { raise Commentaires_mal_balances }
|  _        { traite_normal lexbuf ; normal lexbuf }
|  eof      { () }

```

```

and commentaire = parse
  "(*"      { ignore_lex lexbuf ; commentaire lexbuf ;
|  "*)"     { ignore_lex lexbuf }

```

```

| _      { traite_comment lexbuf ; commentaire le
| eof    { raise Commentaires_mal_balances }

{

let decommente src dest =
  let file_in = open_in src in
  let lb = Lexing.from_channel file_in in
  let file_out = open_out dest in
  init_sortie file_out ;
  normal lb ;
  close_in file_in ;
  close_out file_out ;;

let usage () =
  print_string "comment2 filein fileout";
  print_newline() ;;

let main () =
  if Array.length (Sys.argv) <> 3 then usage ()
  else decommente Sys.argv.(1) Sys.argv.(2) ;;

main ();;
}

```

3. En Objective CAML les chaînes de caractères peuvent contenir n'importe quel caractères, y compris les suites (* ou *). Par exemple la chaîne de caractères "un te(*xte

quelco*)nque" ne doit pas être considéré comme un commentaire. [Modifier](#) l'analyseur lexical pour prendre en compte les chaînes de caractères.

```
(** fichier comment3.mll **)
{
  let ignore_lex lb = ignore (Lexing.lexeme lb)
  let sortie = ref stdout
  let init_sortie f = sortie := f
  let traite_normal lb = output_string !sortie (Lexi
  let traite_comment = ignore_lex
  exception Commentaires_mal_balances
}

rule normal = parse
  "(*"      { ignore_lex lexbuf ; commentaire lexbuf ;
|  "*)"     { raise Commentaires_mal_balances }
|  "'"      { traite_normal lexbuf ; chaine lexbuf ;
|  _        { traite_normal lexbuf ; normal lexbuf }
|  eof      { () }

and commentaire = parse
  "(*"      { ignore_lex lexbuf ; commentaire lexbuf ;
|  "*)"     { ignore_lex lexbuf }
|  _        { traite_comment lexbuf ; commentaire le
|  eof      { raise Commentaires_mal_balances }

and chaine = parse
  "'"      { traite_normal lexbuf ; () }
|  "\\\"    { traite_normal lexbuf ; chaine lexbuf
|  _        { traite_normal lexbuf ; chaine lexbuf }
```

4. Utiliser ce nouvel analyseur pour enlever les commentaires d'un [programme](#) Objective CAML.

```

(** fichier comment4.mll **)
{
  let ignore_lex lb = ignore (Lexing.lexeme lb)
  let sortie = ref stdout
  let init_sortie f = sortie := f
  let traite_normal lb = output_string !sortie (Lexi
  let traite_comment = ignore_lex
  exception Commentaires_mal_balances
}

rule normal = parse
  "(*"      { ignore_lex lexbuf ; commentaire lexbuf ;
  |  "*)"    { raise Commentaires_mal_balances }
  |  "'"     { traite_normal lexbuf ; chaine lexbuf ;
  |  _       { traite_normal lexbuf ; normal lexbuf }
  |  eof     { () }

and commentaire = parse
  "(*"      { ignore_lex lexbuf ; commentaire lexbuf ;
  |  "*)"    { ignore_lex lexbuf }
  |  _       { traite_comment lexbuf ; commentaire le
  |  eof     { raise Commentaires_mal_balances }

and chaine = parse
  "'"       { traite_normal lexbuf ; () }
  |  "\\\"   { traite_normal lexbuf ; chaine lexbuf
  |  _       { traite_normal lexbuf ; chaine lexbuf }

{

let decommente src dest =
  let file_in = open_in src in
  let lb = Lexing.from_channel file_in in
  let file_out = open_out dest in
  init_sortie file_out ;
  normal lb ;
  close_in file_in ;
  close_out file_out ;;

```

```

let usage () =
  print_string "comment2 filein fileout";
  print_newline() ;;

let main () =
  if Array.length (Sys.argv) <> 3 then usage ()
  else decommente Sys.argv.(1) Sys.argv.(2) ;;

main ();;
}

```

Évaluateur

On utilise ocaml yacc pour réaliser un évaluateur d'expressions. L'idée est d'associer directement l'évaluation des expressions aux règles de grammaire.

On choisit un langage d'expressions arithmétiques préfixées (et complètement parenthésées) avec des opérateurs d'arité variable. Par exemple, l'expression (ADD e1 e2 .. en) est équivalente à $e1 + e2 + \dots + en$. Les opérateurs d'addition et de multiplication sont associatifs à droite et ceux de soustraction et de division associatifs à gauche.

1. Définir dans un fichier [open_parser.mly](#) l'analyse syntaxique et les règles d'évaluation d'une expression.

```

%{
  let rec app_right f xs =
    match xs with
    [x] -> x
    | x::xs -> f x (app_right f xs)
    | _ -> failwith "missing argument" ;;
  let rec app_left f xs =

```

```

match xs with
  [x] -> x
  | x1::x2::xs -> app_left f ((f x1 x2)::xs)
  | _ -> failwith"missing argument" ;;

let t = Hashtbl.create 3 ;;

(
  Hashtbl.add t "ADD" (app_right (+.));
  Hashtbl.add t "SUB" (app_left (-.));
  Hashtbl.add t "MUL" (app_right ( *.));
  Hashtbl.add t "DIV" (app_left (/.)
) ;;

let apply o vs =
try
  (Hashtbl.find t o) vs
with
  Not_found -> (Printf.eprintf"Unknown operator %s\n"
%)

%token Lpar Rpar
%token <float> Num
%token <string> Atom

%start term
%type <float> term
%type <float list> terms

%%
term :
  Num                { $1 }
  | Lpar Atom terms Rpar
  { (apply $2 $3) }
;
terms :
  term                { [$1] }
  | term terms        { $1::$2 }
;
%%

```

2. Définir dans un fichier `opn_lexer.mll` l'analyse lexicale des expressions.

```
{
  open Opn_parser
}

rule lexer = parse
  [' ' '\n']           { lexer lexbuf }
| '('                  { Lpar }
| ')'                  { Rpar }
| '-'?['0'-'9']*'.'?['0'-'9']*
  { Num (float_of_string (Lexing.lexeme lexbuf)) }
| ['A'-'z']+          { Atom (Lexing.lexeme lexbuf) }
```

3. Écrire un petit `programme` principal `opn` qui lit une ligne sur l'entrée standard et affiche le résultat de l'évaluation.

```
open Opn_lexer ;;
open Opn_parser ;;

Printf.printf "? "; flush stdout;
let buf = Lexing.from_string (input_line stdin) in
  Printf.printf "= %f\n" (Opn_parser.term Opn_lexer.
```

Résumé

Ce chapitre a décrit les différents outils d'analyses lexicales et syntaxiques d'Objective CAML. On peut citer dans l'ordre d'apparition :

- le module `Str` pour filtrer des expressions rationnelles;
- le module `Genlex` pour construire facilement des analyseurs lexicaux simples;
- l'outil `ocamllex`, intégration typée de l'outil `lex`;

- l'outil ocaml yacc, intégration typée de l'outil yacc;
- les flots pour construire des analyseurs descendants, y compris contextuels.

Les outils ocamllex et ocaml yacc ont été utilisés pour définir une analyse syntaxique du langage Basic plus aisément modifiable que celle présentée page ??.

Pour en savoir plus

L'ouvrage de référence sur les analyses lexicale et syntaxique est connu sous le surnom du « dragon » qui illustre sa couverture. Son vrai titre est *Compilateur : principes, techniques et outils* ([ASU89]). Il couvre toutes les étapes de la compilation. Il explique clairement la construction des automates correspondant à une grammaire non contextuelle donnée et les techniques pour le minimiser. Les outils lex et yacc sont largement décrits dans différents ouvrages, une bonne référence est [LMB92]. L'intérêt de ocamllex et ocaml yacc par rapport à leurs versions originales est l'intégration au langage Objective CAML et surtout de pouvoir construire des analyseurs typés. Sur les *streams*, le rapport de recherche de Michel Mauny et Daniel de Rauglaudre [MdR92] permet de bien comprendre la sémantique opérationnelle de cette extension. D'autre part la lecture de [CM95] montre comment construire une telle extension. Pour une meilleure intégration des grammaires dans le langage Objective CAML, ou pour modifier la grammaire de ce dernier, on peut utiliser l'outil camlp4 que l'on trouve à :

Lien: <http://caml.inria.fr/camlp4/>

Interopérabilité avec C

Introduction

Le développement d'applications dans un langage donné nécessite très souvent de devoir intégrer des bibliothèques provenant d'autres langages. Cela pour deux raisons principales :

- utiliser des bibliothèques impossibles à écrire dans ce langage, c'est-à-dire étendre les fonctionnalités ;
- utiliser des bibliothèques performantes mises en oeuvre dans d'autres langages.

Une application devient alors un assemblage de briques logicielles provenant de différents langages, où chaque brique spécialisée a été écrite dans le langage adéquat pour traiter une partie du problème. Celles-ci interopèrent entre elles pour la communication de valeurs et la demande de calculs.

Le langage Objective CAML possède un tel mécanisme d'interopérabilité avec le langage C. Il permet d'appeler à partir d'Objective CAML des fonctions C, avec arguments, et de récupérer en Objective CAML le résultat du calcul. De même l'inverse est possible. Un programme C peut déclencher un calcul en Objective CAML et travailler ensuite sur le résultat.

Le choix du langage C apparaît judicieux pour les raisons suivantes :

- c'est un langage normalisé (C ANSI) ;
- il est utilisé pour implanter les systèmes d'exploitation (Unix, Windows, MacOS, etc.) ;
- de très nombreuses bibliothèques sont écrites en C ;
- la plupart des langages possèdent une interface avec C, ainsi il sera possible de s'interfacer avec cet autre langage en passant par C.

On peut donc voir le langage C comme un espéranto des langages de programmation.

La coopération entre C et Objective CAML soulève un certain nombre de difficultés que nous passons en revue ci-dessous.

- **représentation mémoire des valeurs**

Par exemple les valeurs de base (*int*, *char*, *float*) des deux langages n'ont pas la même représentation mémoire. Il sera nécessaire de les convertir dans les deux sens. Il en est de même pour les valeurs structurées comme les enregistrements, les types sommes ou les tableaux.

- **GC d'Objective CAML**

Bien qu'il soit possible d'implanter un tel mécanisme en C, celui-ci n'en possède pas dans la définition de sa norme. Il ne faut donc pas que l'appel d'une fonction C fasse des modifications mémoire incompatibles avec le GC d'Objective CAML.

- **ruptures de calcul**

Les deux langages ne possèdent pas le même mécanisme d'exceptions, ce qui introduit un problème pour le récupérateur d'exceptions d'Objective CAML.

- **partage de ressources communes**

Comme par exemple les tampons d'entrées-sorties qui ne sont pas partagés et ne rendent donc pas compte de la séquentialité des entrées-sorties des deux mondes.

Les applications développées en Objective CAML bénéficient des qualités dues à la sûreté du typage et de la gestion automatique de la mémoire. Une bonne utilisation de bibliothèques C ou de l'interfaçage *via* C avec d'autres langages ne doit pas mettre en danger cette sûreté. Il faudra donc suivre des règles assez strictes pour que les deux langages cohabitent harmonieusement.

Plan du chapitre

Ce chapitre présente les outils de la distribution d'Objective CAML qui permettent l'interaction entre Objective CAML et C en créant des exécutables contenant des parties développées dans les deux langages. Ces outils contiennent les fonctions de conversion des valeurs des deux langages, permettent une allocation en C sûre qui utilise le tas Objective CAML et son GC, et autorisent le déclenchement d'exceptions Objective CAML en C.

La première section montre comment utiliser une fonction C en Objective CAML et comment construire des exécutables et des toplevels incluant un programme C contenant cette fonction. La deuxième section explore la représentation des valeurs Objective CAML en C. La troisième section explique comment créer et modifier des valeurs Objective CAML en C. Elle détaille d'autre part les problèmes soulevés par l'allocation mémoire de C en présence du GC d'Objective CAML et comment profiter de ce mécanisme pour

avoir une allocation sûre en C. La quatrième section décrit la gestion des exceptions, leur déclenchement et leur récupération selon le lieu de la rupture de calcul. La cinquième section inverse les rôles en incluant du code Objective CAML dans un programme principal C.

Remarque

La lecture de ce chapitre nécessite de connaître un tant soit peu le langage C. De même la lecture du chapitre 9 est utile pour la compréhension des difficultés liées à la récupération automatique de mémoire.

Communication entre C et Objective CAML

La communication entre parties, d'un même programme, écrites en C et en Objective CAML s'effectue en créant un exécutable (ou une nouvelle boucle d'interaction) contenant ces deux parties. Celles-ci ont pu être compilées séparément. Ce sera donc à l'édition de liens¹ d'établir le lien entre les noms Objective CAML et les noms C, puis de créer l'exécutable. Pour cela le programme Objective CAML contiendra des déclarations externes assurant cette correspondance.

L'exemple de la figure 12.1 montre un programme composé d'une partie C et d'une partie Objective CAML.

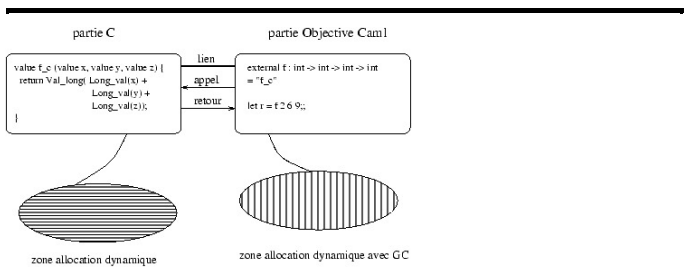


Figure 12.1 : communication entre Objective CAML et C

On peut voir chaque partie comme contenant du code, des instructions correspondant aux définitions de fonctions et de calcul d'expressions pour Objective CAML, ainsi que leur zone d'allocation dynamique de mémoire. L'appel de la fonction *f* sur trois entiers d'Objective CAML entraîne le calcul de la fonction *f_c* de *C*. Le corps de la fonction convertit les trois entiers Objective CAML en entiers *C*, calcule leur somme et retourne le résultat converti en entier Objective CAML.

Nous présentons ci-dessous les premiers éléments d'interfaçage entre Objective CAML et *C* : les déclarations externes, les contraintes sur les définitions de fonctions *C* appelables à partir d'Objective CAML ainsi que les options d'édition de liens. Puis nous donnons un exemple d'utilisation des entrées-sorties.

Déclarations externes

La déclaration externe de fonctions en Objective CAML a pour but d'effectuer la correspondance entre une déclaration de fonction *C* et un nom Objective CAML, tout en indiquant le type de celle-ci.

La syntaxe est la suivante :

Syntaxe

external*nom_caml*:type = "*nom_C*" Cette déclaration indique que l'appel en Objective CAML de la fonction *nom_caml* déclenchera l'appel à la fonction *C* *nom_C* avec ses arguments. L'exemple de la figure 12.1 déclare ainsi la fonction *f* dont l'appel correspondra à l'exécution de la fonction *f_c*.

Il est possible de déclarer une fonction externe dans une interface (i.e. dans un fichier *.mli*), soit en la déclarant explicitement externe, soit comme une valeur usuelle :

Syntaxe

external*nom_caml:type = "nom_C"***val***nom_caml:type* Dans le second cas, l'appel à la fonction C passe au préalable par le mécanisme général des fonctions Objective CAML. C'est un peu moins efficace mais cache la nature de l'implantation de la fonction.

Déclaration des fonctions C

Les fonctions C appelables à partir d'Objective CAML doivent posséder le nombre d'arguments décrit dans la déclaration externe. Ces arguments sont de type *value* qui est, en C, le type des valeurs d'Objective CAML. Comme celles-ci ont une représentation uniforme (9) un seul type C permet de coder toutes les valeurs Objective CAML. Nous exposerons page ?? l'utilisation des outils de codage-décodage des valeurs et l'illustrerons par une fonction d'exploration des valeurs d'Objective CAML.

L'exemple de la figure 12.1 est conforme à ces contraintes. La fonction *f_c*, liée à la fonction Objective CAML de type *int -> int -> int -> int* est bien une fonction à trois paramètres de type *value* qui retourne un résultat de type *value*.

Le mécanisme d'évaluation de l'interprète de code-octet d'Objective CAML diffère selon le nombre d'arguments². Si le nombre d'arguments est inférieur ou égal à cinq, les arguments sont empilés dans la pile d'exécution de la machine. Si le nombre d'arguments est supérieur à cinq, c'est un tableau contenant tous les arguments qui est empilé ainsi que la taille de ce tableau. Comme les fonctions C peuvent être appelées à partir de la machine abstraite, il est nécessaire de distinguer ces deux cas. Par contre l'appel de fonction à partir du compilateur natif empile bien tous ses arguments. Ceux-ci pourront donc être passés directement à la fonction C.

Cas des fonctions à plus de cinq arguments

Il convient dans ce cas d'écrire deux fonctions C : l'une pour le code-octet et l'autre pour le code natif. La syntaxe des déclarations externes est donc étendue pour permettre une seule déclaration de fonction Objective CAML pour deux fonctions C :

Syntaxe

external*nom_caml*:type = "*nom_C_byte_code*" "*nom_C_natif*" La fonction *nom_C_byte_code* prend deux arguments : un tableau de valeurs de type *value* (i.e. un pointeur de type **value*) et un entier indiquant la taille de ce tableau.

Exemple

Le programme C suivant définit deux fonctions différentes pour l'addition de six entiers : *plus_native* et *plus_bytecode* qui seront utilisées respectivement par le compilateur natif et le compilateur de code-octet. Il est nécessaire d'inclure le fichier *mlvalues.h* contenant les définitions des types C des valeurs Objective CAML et les macros de conversion.

```
#include <stdio.h>
#include <caml/mlvalues.h>

value plus_native (value x1,value x2,value x3,value x4,value x5,value x6)
{
    printf("<< PLUS NATIF >>\n") ; fflush(stdout) ;
    return Val_long ( Long_val(x1) + Long_val(x2) + Long_val(x3) +
+ Long_val(x4) + Long_val(x5) + Long_val(x6)) ;
}

value plus_bytecode (value * tab_val, int nb_val)
{
    int i;
    long res;
    printf("<< PLUS BYTECODE >> : ") ; fflush(stdout) ;
    for (i=0,res=0;i<nb_val;i++) res += Long_val(tab_val[i]);
    return Val_long(res) ;
}
```

Le programme Objective CAML *exOCAML.ml* suivant utilise ces fonctions C.

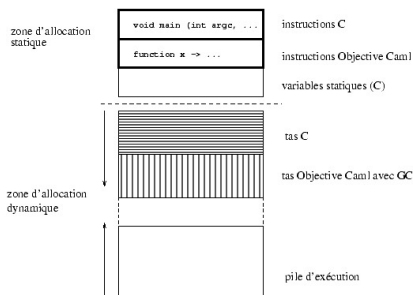


Figure 12.2 : exécutable mixte

Les instructions machine provenant de la compilation des programmes C et Objective CAML sont rangées dans la zone d'allocation statique du programme. La zone d'allocation dynamique contient la pile d'exécution (où sont rangés les appels en cours) et les tas de C et d'Objective CAML.

Bibliothèque d'exécution

Les primitives C susceptibles d'être appelées par un programme n'utilisant que les bibliothèques standard sont contenues dans la bibliothèque d'exécution de la machine abstraite (voir la figure 7.3 page ??). Il n'est donc pas nécessaire de préciser quoi que ce soit. En revanche, lorsqu'on utilise les bibliothèques Graphics, Num ou Str il est nécessaire d'explicitement les bibliothèques correspondant à ces modules pour l'édition de lien. C'est la fonction de l'option `-custom` du compilateur. Il en va de même pour des fonctions C que nous souhaiterions utiliser, il faut ajouter le fichier objet contenant ces fonctions au moment de l'édition de lien. Nous allons détailler cela dans un exemple.

Différents cadres d'édition de liens

Il faut distinguer les cadres de compilation pour le compilateur natif, le compilateur de code-octet et la construction de boucles

d'interaction. Les différentes options des différents modes de compilation sont décrites au chapitre 7.

Pour illustrer ces différents modes, on reprend l'exemple de la figure 12.1 en nommant le fichier Objective CAML `progocaml.ml`. Celui-ci utilise la fonction externe `f_c` définie dans le fichier C `progC.c`. Ce dernier utilise une bibliothèque C `libC`. Une fois ces fichiers compilés de manière indépendante, leur édition de liens s'effectue par les commandes suivantes :

- **code-octet :**
`ocamlc -custom -o vbc.exe progC.o libC.a progocaml.cmo`
- **natif :**
`ocamlopt progC.o -o vn.exe libC.a progocaml.cmx`

On obtient deux exécutables : `vbc.exe` pour la version en code-octet et `vn.exe` pour la version en code natif.

Construire une nouvelle machine abstraite

On peut aussi augmenter la bibliothèque d'exécution de la machine abstraite en intégrant de nouvelles fonctions C. Pour cela on utilisera les commandes suivantes :

```
ocamlc -make-runtime -o nouveau_ocamlrun progC.o libC.a
```

On pourra alors construire un exécutable non autonome, `vbcnam.exe`, utilisant la nouvelle machine abstraite :

```
ocamlc -o vcam.exe -use-runtime nouveau_ocamlrun progC.o  
libC.a progocaml.cmo
```

Il sera exécuté par la commande : `nouveau_ocaml vbcam.exe`,
ou directement par : `vbcam.exe`

Remarque

L'édition de liens en mode -custom force le compilateur à parcourir les fichiers objets (.cmo) pour y répertorier les fonctions externes mentionnées. Le code-octet nécessaire à leur utilisation est engendré et ajouté au code-octet provenant des programmes Objective CAML.

Construire une boucle d'interaction

Pour pouvoir utiliser une fonction externe dans une boucle d'interaction, il faut en construire une nouvelle comportant d'une part le code C de la fonction et d'autre part un programme Objective CAML contenant sa déclaration.

On suppose compilé le fichier progC.c contenant la fonction f_c. On construit alors la boucle d'interaction ftop de la manière suivante :

```
ocamlmktop -custom -o ftop progC.o libC.a ex.ml
```

Le fichier ex.ml contient la déclaration externe de la fonction f. Le toplevel ftop connaît alors cette fonction et dispose du code C correspondant qui est donné dans progC.o.

Entrées-sorties des deux mondes

Les fonctions C et Objective CAML ne partagent pas leurs tampons d'échange avec les fichiers. Soit le programme C suivant :

```
#include <stdio.h>
#include <caml/mlvalues.h>
value hello_world (value v)
{ printf("Hello World !!");  fflush(stdout);  return v;
```

On note que les écritures sur la sortie standard doivent être forcées (fflush) si l'on souhaite qu'elles apparaissent dans le bon ordre.

```
# external caml_hello_world : unit -> unit = "hello_world"
external caml_hello_world : unit -> unit = "hello_world"
# print_string "<< " ;
```

```

caml_hello_world () ;
print_string " >>\n" ;
flush stdout ;;
Hello World !!<<  >>
- : unit = ()

```

Cet affichage n'est pas satisfaisant. Il faut réécrire le programme Objective CAML de la manière suivante :

```

# print_string "<< " ; flush stdout ;
caml_hello_world () ;
print_string " >>\n" ; flush stdout ;;
<< Hello World !! >>
- : unit = ()

```

Le vidage des tampons qui suit systématiquement un ordre d'écriture permet de conserver l'ordre d'affichage entre les deux mondes.

Exploration des valeurs Objective CAML en C

Le langage Objective CAML ne construit pas les valeurs de la même façon que le langage C, y compris pour les valeurs simples comme les entiers. Cela provient de la nécessité pour le GC d'Objective CAML de conserver des informations supplémentaires pour effectuer la récupération mémoire. Comme la représentation des valeurs en Objective CAML est uniforme, elles seront vues en C comme des valeurs d'un type unique, nommé `value`.

Quand Objective CAML appelle une fonction C et lui passe des arguments, ceux-ci doivent être décodés pour être utilisés en C. Il en est de même pour le résultat d'une fonction C appelée depuis Objective CAML. Celui-ci sera encodé avant d'être transmis à Objective CAML. Un certain nombre de macros et de fonctions C réalisent ces opérations de conversion de valeurs. Elles sont regroupées dans les fichiers décrits à la figure [12.3](#) qui sont fournis dans la distribution d'Objective CAML. Ces fichiers se trouvent dans le répertoire `LIBOCAML/caml` où `LIBOCAML` est le répertoire où les bibliothèques d'Objective CAML sont installées³.

mlvalues.h	définition du type <i>value</i> et macros de manipulations de ces valeurs.
alloc.h	fonctions d'allocations de valeurs Objective CAML.
memory.h	macros de plus haut niveau pour la manipulation de valeurs.

Figure 12.3 : Fichiers utiles pour l'interfaçage avec C

Classification des valeurs de type

Une valeur de type *value* peut être soit :

- un entier ;
- un pointeur vers le tas d'Objective CAML ;
- un pointeur hors du tas d'Objective CAML.

Le tas d'Objective CAML est la zone d'allocation gérée par le GC d'un programme Objective CAML. Un programme C, peut pour sa part créer et manipuler des valeurs dans sa propre zone d'allocation et communiquer une référence de cette valeur à Objective CAML.

On dispose des macros de vérification et de conversion de types décrites à la figure [12.4](#).

Is_long(v)	v est un entier Objective CAML?
Is_block(v)	v est un pointeur Objective CAML?

Long_val(v)	retourne un entier C long
Int_val(v)	retourne un entier C
Bool_val(v)	retourne un ``booléen" C (0 pour false)

Figure 12.4 : Conversion des valeurs immédiates

Il est à noter que C possède plusieurs types pour les entier : short, int et long alors qu'Objective CAML n'a qu'une seule représentation des entiers : le type int.

Accès aux valeurs immédiates

Les entiers servent à encoder toutes les valeurs immédiates d'Objective CAML :

- les entiers sont codés par leur valeur,
- les caractères sont codés par leur code ASCII,
- les constructeurs constants sont codés par un entier dénotant leur place dans la déclaration du type : le n^{ème} constructeur constant d'un type énuméré sera codé par $n-1$.

Le programme suivant définit une fonction C inspecte d'inspection une valeur de type value :

```
#include <stdio.h>
#include <caml/mlvalues.h>
value inspecte (value v)
{
    if (Is_long(v))
        printf ("v est un entier (%d) : %d", (int) v, Long_val(v))
    else if (Is_block(v)) printf ("v est un pointeur") ;
    else printf ("v n'est ni un entier ni un pointeur ???")
    printf("    ");
    fflush(stdout) ;
    return v ;
}
```

La fonction `inspecte` teste si son argument est un entier Objective CAML. Si oui elle en affiche la valeur vue comme un `int C` puis cette même valeur convertie par la macro `Long_val` en entier `C` (long).

La représentation des entiers d'Objective CAML diffère de celle de `C` comme le montre l'utilisation suivante de la fonction `inspecte`.

```
# external inspecte : 'a -> 'a = "inspecte" ;;
external inspecte : 'a -> 'a = "inspecte"
# inspecte 123 ;;
v est un entier (247) : 123    - : int = 123
# inspecte max_int;;
v est un entier (2147483647) : 1073741823    - : int = 1073741823
```

D'autres valeurs de type prédéfinis, comme *char* et *bool*, sont représentées par des valeurs immédiates.

```
# inspecte 'A' ;;
v est un entier (131) : 65    - : char = 'A'
# inspecte true ;;
v est un entier (3) : 1      - : bool = true
# inspecte false ;;
v est un entier (1) : 0      - : bool = false
# inspecte [] ;;
v est un entier (1) : 0      - : '_a list = []
```

Soit le type `foo` défini en Objective CAML :

```
# type foo = C1 | C2 of int | C3 | C4 ;;
```

La fonction `inspecte` affichera des résultats de nature différente en présence d'un constructeur constant ou d'un constructeur fonctionnel.

```
# inspecte C1 ;;
v est un entier (1) : 0      - : foo = C1
# inspecte C4 ;;
v est un entier (5) : 2      - : foo = C4
```

```
# inspecte (C2 1) ;;
v est un pointeur    - : foo = C2 1
```

Lorsqu'elle reconnaît une valeur immédiate, la fonction `inspecte` affiche entre parenthèses le contenu « physique » de cette valeur (i.e. sa valeur vue comme un entier signé sur un mot -- type `int` de C), puis elle affiche son contenu « logique » (la valeur obtenue par les macros de décodage). Nos exemples manifestent la différence entre ces deux contenus. Celle-ci provient du *bit de tag*⁴ qu'utilise le GC pour distinguer une valeur immédiate d'un pointeur (voir chapitre 9, page ??).

Discrimination des valeurs structurées

Les valeurs structurées d'Objective CAML correspondent à toutes les valeurs non immédiates : n-uplets, listes non vides, constructeurs avec au moins un argument, vecteurs, enregistrements, fermetures et valeurs de type abstrait. Elles sont allouées dans le tas Objective CAML sous forme de blocs mémoire. Un bloc possède un en-tête contenant une information sur la nature du bloc ainsi que sa taille en mots machine. La figure 12.5 décrit la structure d'un bloc pour une machine à mots de 32 bits.

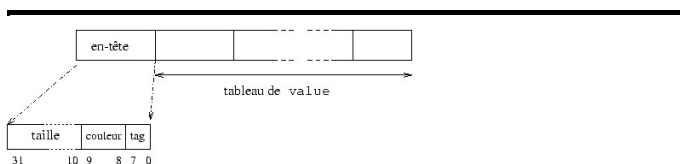


Figure 12.5 : communication de valeurs entre Objective CAML et C

Les deux bits « couleur » sont utilisés par le GC lors de son exploration du graphe des valeurs du tas (voir chapitre 9, page ??). Le champ « tag » (que nous appellerons *tag*, pour faire court) indique le « type » de valeurs contenues dans la suite du bloc. Les fonctions de la figure 12.6 retournent ces informations.

Wosize_val(v)	retourne la taille du bloc (sans l'en-tête)
Tag_val(v)	retourne le <i>tag</i> du bloc

Figure 12.6 : Informations sur les blocs mémoire

Les différentes valeur de *tag* sont données à la figure 12.7.

de 0 à No_scan_tag-1	un tableau Objective CAML: un tableau de value
Closure_tag	une fermeture
String_tag	une chaîne de caractères
Double_tag	un flottant en double- précision
Double_array_tag	un tableau de flottants
Abstract_tag	un type de données abstrait
Final_tag	idem avec une fonction de finalisation

Figure 12.7 : Description des marques des blocs mémoire

Suivant la valeur retournée par Tag_val, on utilise différentes macros pour accéder aux contenus des blocs mémoire. Lorsque cette valeur est inférieure à No_scan_tag, le bloc mémoire est structuré comme un tableau de value. Les macros d'accès sont décrites à la figure 12.8. En accord avec les conventions de C et d'Objective CAML le premier élément d'un tableau se situe à l'indice 0.

Field(v,n)	retourne la n ^{ème} value
------------	------------------------------------

<code>Code_val(v)</code>	d'un tableau. retourne le pointeur de code d'une fermeture.
<code>string_length(v)</code>	retourne la longueur d'une chaîne.
<code>Byte(v,n)</code>	retourne le n ^{ème} caractère d'une chaîne comme char.
<code>Byte_u(v,n)</code>	idem mais retourne un unsigned char.
<code>String_val(v)</code>	retourne la chaîne de caractère comme (char *).
<code>Double_val(v)</code>	retourne le flottant codé dans v.
<code>Double_field(v,n)</code>	retourne le n ^{ème} élément d'un tableau de flottants.

Figure 12.8 : Accès aux éléments d'un bloc

Comme nous l'avons fait pour les valeurs immédiates, nous allons définir une fonction d'inspection des blocs mémoires. La fonction `C_print_bloc` teste si son argument (de type `value`) est une valeur immédiate ou un bloc mémoire. Dans ce dernier cas elle affiche le type du bloc et sa valeur vue de C. Elle est appelée depuis `inspecte_bloc` qui sera utilisée à partir d'un programme Objective CAML.

```
#include <stdio.h>
#include <caml/mlvalues.h>

void marge (int n)
{ for (;n>0;n--) printf("."); return; }

void print_bloc (value v,int m)
{
  int taille,i ;
```

```

marge(m);
if (Is_long(v))
{ printf("valeur immediate (%d)\n", Long_val(v)); return; }
printf ("bloc : taille=%d - ", taille=Wosize_val(v));
switch (Tag_val(v))
{
case Closure_tag :
printf("fermeture - Arguments (%d) :\n",taille==1?0:taille);
marge(m+4); printf("pointeur de code : %d\n",Code_val(v));
for (i=1;i<taille;i++) print_bloc(Field(v,i), m+4);
break;
case String_tag :
printf("string : %s (%s)\n", String_val(v),(char *) v);
break;
case Double_tag:
printf("flottant : %g\n", Double_val(v));
break;
case Double_array_tag :
printf ("tableau de flottants : ");
for (i=0;i<(taille/2);i++) printf(" %g", Double_field(v,i));
printf("\n");
break;
case Abstract_tag : printf("type abstrait\n"); break;
case Final_tag : printf("type abstrait + final\n"); break;
default:
if (Tag_val(v)>=No_scan_tag) { printf("inconnu"); break; }
printf("tableau de valeurs (Tag=%d) :\n",Tag_val(v));
for (i=0;i<taille;i++) print_bloc(Field(v,i),m+4);
}
return ;
}

```

```

value inspecte_bloc (value v)
{ print_bloc(v,4); fflush(stdout); return v; }

```

Les différentes marques de blocs se retrouvent dans les cas du switch. On redéfinit ensuite une fonction inspecte :

```
# external inspecte : 'a -> 'a = "inspecte_bloc" ;;
external inspecte : 'a -> 'a = "inspecte_bloc"
```

Celle-ci est utilisée pour décrire expérimentalement les représentations des valeurs structurées Objective CAML. On évitera de lui passer une structure circulaire car son parcours bouclerait.

Tableaux, n-uplets et enregistrements

Les tableaux (*array*) et les n-uplets sont codés par des tableaux de value.

```
# inspecte [| 1; 2; 3 |] ;;
....bloc : taille=3 - tableau de valeurs (Tag=0) :
.....valeur immediate (1)
.....valeur immediate (2)
.....valeur immediate (3)
- : int array = [|1; 2; 3|]
# inspecte ( 10 , true , () ) ;;
....bloc : taille=3 - tableau de valeurs (Tag=0) :
.....valeur immediate (10)
.....valeur immediate (1)
.....valeur immediate (0)
- : int * bool * unit = 10, true, ()
```

Les enregistrements sont eux aussi codés par des tableaux de value dans le même ordre que celui de la déclaration du type. Le fait qu'un champ soit modifiable ne change pas sa représentation physique.

```
# type foo = { ch1 : int ; mutable ch2:int } ;;
type foo = { ch1: int; mutable ch2: int }
# inspecte { ch1=10 ; ch2=20 } ;;
....bloc : taille=2 - tableau de valeurs (Tag=0) :
.....valeur immediate (10)
.....valeur immediate (20)
- : foo = {ch1=10; ch2=20}
```

Warning

Rien n'empêche de modifier physiquement un champ non modifiable d'un enregistrement Objective CAML à partir de fonctions C. C'est au programmeur de prendre garde à ne pas introduire d'incohérence en utilisant une fonction C.

Types somme

Nous avons vu précédemment que les constructeurs constants sont représentés par des entiers. Les autres constructeurs sont codés par le tableau de leurs arguments. Le constructeur est identifié par la valeur de *tag*. Il représente le numéro d'apparition du constructeur non constant dans la déclaration du type : 0 correspond au premier constructeur non constant, 1 au deuxième et ainsi de suite.

```
# type foo = C1 of int * int * int | C2 of int | C3 | C4
type foo = | C1 of int * int * int | C2 of int | C3 | C4
# inspecte (C1 (1,2,3)) ;;
....bloc : taille=3 - tableau de valeurs (Tag=0) :
.....valeur immediate (1)
.....valeur immediate (2)
.....valeur immediate (3)
- : foo = C1 (1, 2, 3)
# inspecte (C4 (1,2)) ;;
....bloc : taille=2 - tableau de valeurs (Tag=2) :
.....valeur immediate (1)
.....valeur immediate (2)
- : foo = C4 (1, 2)
```

Remarque

Le type *list* est un type somme dont la déclaration est `:type 'a list = [] | :: of 'a * 'a list`. Il possède un seul constructeur non constant infixe (`::`) qui a un *tag* égal à 0.

Chaînes de caractères

Comme un caractère est codé sur un octet, la représentation des chaînes de caractères utilisera un mot pour quatre caractères.

Warning

Les chaînes de caractères Objective CAML peuvent contenir le caractère de code ASCII 0 qui est le caractère de fin de chaîne en C.

```
#include <stdio.h>
#include <caml/mlvalues.h>

value explore_string (value v)
{
  char *s;
  int i,taille;
  s = (char *) v;
  taille = Wosize_val(v) * sizeof(value);
  for (i=0;i<taille;i++)
  {
    int p = (unsigned int) s[i] ;
    if ((p>31) && (p<128)) printf("%c",s[i]) ;
    else printf("(#%u)",p) ;
  }
  printf("\n");
  fflush(stdout);
  return v;
}
```

La fin d'une chaîne Objective CAML est déterminée par la taille du bloc la constituant, ainsi que par le dernier octet du dernier mot du bloc qui indique le nombre d'octets *non utilisés* dans le dernier mot. Les exemples ci-dessous permettent de comprendre le rôle exact que joue ce dernier octet.

```
# external explore : string -> string = "explore_string"
external explore : string -> string = "explore_string"
# ignore(explore "");
  ignore(explore "a");
```

```

ignore(explore "ab");
ignore(explore "abc");
ignore(explore "abcd");
ignore(explore "abcd\000") ;;
(#0) (#0) (#0) (#3)
a(#0) (#0) (#2)
ab(#0) (#1)
abc(#0)
abcd(#0) (#0) (#0) (#3)
abcd(#0) (#0) (#0) (#2)
- : unit = ()

```

Dans les deux derniers cas ("abcd" et "abcd\000") les chaînes explorées sont respectivement de longueur 4 et 5. C'est pour cela que le dernier octet change de valeur pour ces deux valeurs.

Flottants et tableaux de flottants

Objective CAML ne possède qu'un seul type (*float*) pour les nombres à virgule flottante. Les valeurs de type *float* sont allouées, elles sont représentées par un bloc de taille 2.

```

# inspecte 1.5 ;;
....bloc : taille=2 - flottant : 1.5
- : float = 1.5
# inspecte 0.0;;
....bloc : taille=2 - flottant : 0
- : float = 0

```

Les tableaux de flottants possèdent une structure spéciale qui permet d'optimiser leur occupation mémoire. C'est pourquoi ils ont un *tag* distinct de celui des autres tableaux ainsi qu'une macro d'accès particulière.

```

# inspecte [| 1.5 ; 2.5 ; 3.5 |] ;;
....bloc : taille=6 - tableau de flottants : 1.5 2.5 3.5
- : float array = [|1.5; 2.5; 3.5|]

```

Cette optimisation permet d'utiliser Objective CAML pour effectuer des calculs numériques qui manipulent ce genre de données beaucoup plus rapidement que s'il fallait passer par un pointeur dans le tas.

Warning

Si l'on alloue un bloc en C pour un tableau de flottants Objective CAML, il sera nécessaire d'allouer un bloc de taille double du nombre d'éléments voulus.

En dehors des *float array*, les flottants contenus dans d'autres structures conservent leur représentation habituelle, c'est-à-dire qu'ils sont vus comme une valeur structurée allouée dans le tas. L'exemple suivant inspecte une liste de flottants.

```
# inspecte [ 3.14; 1.2; 7.6];;
....bloc : taille=2 - tableau de valeurs (Tag=0) :
.....bloc : taille=2 - flottant : 3.14
.....bloc : taille=2 - tableau de valeurs (Tag=0) :
.....bloc : taille=2 - flottant : 1.2
.....bloc : taille=2 - tableau de valeurs (Tag=
.....bloc : taille=2 - flottant : 7.6
.....valeur immediate (0)
- : float list = [3.14; 1.2; 7.6]
```

La liste est vue comme un bloc de taille 2 : sa tête et sa queue. La tête est un flottant, bloc de taille 2 aussi.

Fermetures

Une valeur fonctionnelle est représentée d'une part par le code qui doit être exécuté à son application, et d'autre part par son environnement (voir chapitre 2, page??). Il y a deux façons d'obtenir une valeur fonctionnelle : en utilisation explicitement une abstraction (comme dans *fun x -> x + 1* ou en appliquant partiellement une fonction (comme dans (*fun x -> fun y -> x + y*) 1).

L'environnement d'une fermeture peut contenir trois catégories de variables : celles déclarées globalement, celles déclarées localement et les paramètres instanciés par une application partielle. Du point de vue implantation, ces trois catégories de variables ont un traitement différent. Les variables globales font partie d'un environnement global et ne figurent pas explicitement dans les fermetures. Les variables locales et les paramètres instanciés par application partielle peuvent apparaître dans la fermeture comme nous allons l'illustrer. Ainsi, du point de vue implantation, l'environnement d'une fermeture concerne que des variables locales ou des paramètres abstraits.

Si la fermeture a un environnement vide, elle est uniquement constituée du pointeur vers le code :

```
# let f = fun x y z -> x+y+z ;;
val f : int -> int -> int -> int = <fun>
# inspecte f ;;
....bloc : taille=1 - fermeture - Arguments (0) :
.....pointeur de code : 134893716
- : int -> int -> int -> int = <fun>
```

Si elle est obtenue par application partielle d'une abstraction sans déclaration locale, elle contient une valeur pour chacun des paramètres et une fermeture sans environnement.

```
# let a1 = f 1 ;;
val a1 : int -> int -> int = <fun>
# inspecte (a1) ;;
....bloc : taille=3 - fermeture - Arguments (1) :
.....pointeur de code : 134893712
.....bloc : taille=1 - fermeture - Arguments (0) :
.....pointeur de code : 134893716
.....valeur immediate (1)
- : int -> int -> int = <fun>
# let a2 = a1 2 ;;
val a2 : int -> int = <fun>
# inspecte (a2) ;;
....bloc : taille=4 - fermeture - Arguments (2) :
.....pointeur de code : 134893712
```



```

.....bloc : taille=1 - fermeture - Arguments (0) :
.....pointeur de code : 134893716
.....valeur immediate (1)
.....valeur immediate (2)
- : int -> int = <fun>

```

La figure 12.9 illustre graphiquement l'affichage précédent.

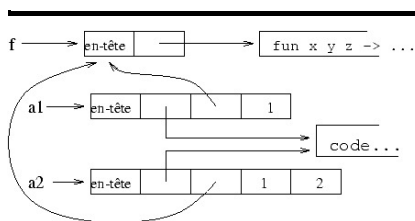


Figure 12.9 : représentation des fermetures

La fonction `f` ne contient pas de variable libre. Le pointeur de code d'une fermeture sans environnement pointe vers le code à appliquer lorsque tous ses arguments sont passés, c'est-à-dire ici, le code correspondant à `x+y+z`. Une fermeture avec environnement pointe vers un code unique (c'est le même pour `a1` et `a2`). Ce code vérifie quand on lui passe un nouvel argument si tous les arguments sont là. Si oui il empile les arguments et déclenche le code initial, sinon il crée une nouvelle fermeture. C'est ce qui se passe quand `a1` est appliquée à `2`. On peut remarquer qu'une fermeture avec environnement contient toujours, comme premier élément de l'environnement, le pointeur vers la fermeture sans environnement dont l'application partielle est issue. Cela permettra, d'une part, de déclencher le code effectif et d'autre part, d'avoir une référence sur soi même pour les fonctions récursives.

En présence d'une déclaration locale, l'application partielle donne la représentation suivante :

```
# let g x = let y=2 in fun z -> x+y+z ;;
```

```

val g : int -> int -> int = <fun>
# let a1 = g 1 ;;
val a1 : int -> int = <fun>
# inspecte a1 ;;
....bloc : taille=3 - fermeture - Arguments (1) :
.....pointeur de code : 134893792
.....valeur immediate (1)
.....valeur immediate (2)
- : int -> int = <fun>

```

L'appel d'une fermeture C à partir d'Objective CAML est détaillé à la section [12](#).

Types abstraits

Une valeur d'un type abstrait est aussi représentée par un tableau de value. En fait l'information sur le type n'est utile que pour le synthétiseur de types. À aucun moment de l'exécution d'un programme l'information de type n'est utile, seule la représentation mémoire doit être connue du GC ainsi que la taille de la valeur.

Si on crée une valeur d'un type abstrait, comme *'a Stack.t*, en fait sa représentation mémoire correspond à celle de son implantation, c'est-à-dire ici comme une référence sur une liste.

```

# let p = Stack.create();;
val p : '_a Stack.t = <abstr>
# Stack.push 3 p;;
- : unit = ()
# inspecte p;;
....bloc : taille=1 - tableau de valeurs (Tag=0) :
.....bloc : taille=2 - tableau de valeurs (Tag=0) :
.....valeur immediate (3)
.....valeur immediate (0)
- : int Stack.t = <abstr>

```

Par contre certains types abstraits sont implantés en utilisant une représentation non exprimable en Objective CAML. C'est par exemple le cas pour les vecteurs de pointeurs faibles ou les canaux

d'entrées-sorties. Pour ceux-là la marque `Abstract_tag` sera utilisée.

```
# let w = Weak.create 10;;
val w : '_a Weak.t = <abstr>
# Weak.set w 0 (Some p);;
- : unit = ()
# inspecte w;;
....bloc : taille=11 - type abstrait
- : int Stack.t Weak.t = <abstr>
```

Certains de ces types possèdent aussi une fonction de finalisation, permettant une dernière action avant de disparaître (en particulier la libération de la mémoire qu'ils occupent). C'est particulièrement utile pour rendre une ressource externe, comme un tampon d'entrées-sorties. C'est pour cela que l'inspection de la sortie standard indiquera que le type `out_channel` possède une fonction de finalisation :

```
# inspecte (stdout) ;;
....bloc : taille=2 - type abstrait + final
- : out_channel = <abstr>
```

Créer et modifier des valeurs Objective CAML en C

L'appel d'une fonction C à partir d'Objective CAML peut modifier ses arguments ou retourner une nouvelle valeur. Cette valeur doit correspondre à un type Objective CAML. Pour les types simples, plusieurs macros C sont prédéfinies pour faciliter les conversions. Pour une valeur d'un type structuré, elle doit être allouée de la bonne taille et ses champs remplis par des valeurs du bon type. Il est en effet facile de faire n'importe quoi en C, ce qui risque de provoquer un arrêt de l'exécution du programme Objective CAML appelant ce morceau de C.

Il est important que les nouvelles valeurs Objective CAML créées en C soient connues du GC d'Objective CAML. Ces valeurs, le temps de leur portée, doivent être considérées comme de nouvelles racines pour le GC. Pour cela plusieurs macros sont définies pour ajouter

des racines à conserver par le GC.

De plus, il est possible d'allouer à partir de C des valeurs Objective CAML mais aussi C, dans le tas d'Objective CAML. Cela permet de profiter du mécanisme de récupération automatique de mémoire d'Objective CAML dans les programmes C. Pour des valeurs dont le type n'est connu qu'en C, il est nécessaire d'indiquer au GC leur fonction de finalisation pour la libération d'un bloc de ce type.

Modification des valeurs Objective CAML

Valeurs immédiates

Les macros suivantes permettent de créer des valeurs immédiates de type `value` à partir de valeur C ainsi que d'en modifier:

<code>Val_long(l)</code>	à partir d'un long int
<code>Val_int(i)</code>	à partir d'un int
<code>Val_bool(x)</code>	à partir d'un int (false si $x=0$, true sinon)
<code>Val_true</code>	
<code>Val_false</code>	
<code>Val_unit</code>	
<code>Store_field(b,n,v)</code>	met la valeur <code>v</code> dans le champ <code>n</code> d'un bloc <code>b</code>
<code>Store_double_field(b,n,d)</code>	idem pour un tableau de flottants

Figure 12.10 : Création et modification de valeurs immédiates

Il est en outre possible d'utiliser l'une des macros `Field`, `Byte` et `Byte_u` pour affecter les champs d'un bloc :

`Store_field(b,n,v) ⇔ Field(b,n) = v.`

Voici un exemple pour inverser une chaîne de caractères :

```
#include <caml/mlvalues.h>
value swap_char(value v, int i, int j)
{ char c=Byte(v,i); Byte(v,i)=Byte(v,j); Byte(v,j)=c; }
value swap_string (value v)
{
  int i,j,t = string_length(v) ;
  for (i=0,j=t-1; i<t/2; i++,j--)  swap_char(v,i,j) ;
  return v ;
}

# external miroir : string -> string = "swap_string" ;;
external miroir : string -> string = "swap_string"
# miroir "abcdefg" ;;
- : string = "gfedcba"
```

Allocation de nouveaux blocs

Il est important d'utiliser les macros suivantes pour allouer des valeurs dans le tas d'Objective CAML, sans cela, les blocs n'auraient pas les informations nécessaires au GC pour les manipuler. La figure [12.11](#) liste les différentes fonctions d'allocation de blocs dans le tas Objective CAML.

<code>alloc(n, t)</code>	retourne un nouveau bloc de taille <code>n</code> et de Tag <code>t</code>
<code>alloc_tuple(n)</code>	idem mais avec le Tag 0
<code>alloc_string(n)</code>	retourne une chaîne (non initialisée) de longueur <code>n</code>
<code>copy_string(s)</code>	retourne un bloc à partir d'une chaîne <code>C</code>
<code>copy_double(d)</code>	retourne un bloc à partir d'un double
<code>alloc_array(f, a)</code>	retourne un bloc tableau à partir d'une fonction <code>f</code>

<code>copy_string_array(p)</code>	de conversion et d'un tableau de pointeurs retourne un tableau de chaîne à partir d'un (char**), le tableau p doit se finir par un pointeur nul
-----------------------------------	---

Figure 12.11 : Fonctions d'allocation de blocs

La macro `alloc_array` prend un tableau de pointeurs (a) se terminant par le pointeur nul et une fonction f qui prend un pointeur et rend une valeur. Elle rend un tableau de valeur obtenu en appliquant f à chacun des pointeurs de a. La fonction `make_str_array` de l'exemple ci-dessous utilise la macro `alloc_array` pour créer des copies de chacune des chaînes d'un tableau :

```
#include <caml/mlvalues.h>
value make_str (char *s) { return copy_string(s); }
value make_str_array (char **p) { return alloc_array(make_str, p, 0); }
```

Il est aussi possible de définir des blocs de taille 0, c'est par exemple le cas d'un tableau Objective CAML vide. Un tel bloc est appelé un *atome*.

```
# inspecte [| |] ;;
....bloc : taille=0 - tableau de valeurs (Tag=0) :
- : '_a array = [| |]
```

Il est possible de l'invoquer en C par la macro `Atom(t)` où t est le tag du bloc créé. En fait, de tels blocs sont constants et ne sont pas alloués dans la partie dynamique du tas Objective CAML.

Des valeurs C dans le tas Objective CAML

On peut souhaiter conserver dans le tas Objective CAML des valeurs qui ne proviennent pas de ce langage et qui ne sont donc pas soumis aux contraintes imposées par Objective CAML pour le GC. Dans ce cas, on utilisera les *blocs abstraits*.

L'exemple le plus naturel est de manipuler avec Objective CAML des entiers C utilisant 32 (ou 64) bits. Une telle valeur va être conservée dans un bloc de taille 1 (la taille d'une valeur est celle d'un entier long) de tag `Abstract_tag`.

```
#include <caml/mlvalues.h>
#include <stdio.h>
```

```
value Cint_of_OCAMLint (value v)
{
  value res = alloc(1,Abstract_tag) ;
  Field(res,0) = Long_val(v) ;
  return res ;
}
```

```
value OCAMLint_of_Cint (value v) { return Val_long(Field(v,0)) ; }
```

```
value Cplus (value v1,value v2)
{
  value res = alloc(1,Abstract_tag) ;
  Field(res,0) = Field(v1,0) + Field(v2,0) ;
  return res ;
}
```

```
value printCint (value v)
{
  printf ("%d", (long) Field(v,0)) ; fflush(stdout) ;
  return Val_unit ;
}
```

```
# type cint
external cint_of_int : int -> cint = "Cint_of_OCAMLint"
external int_of_cint : cint -> int = "OCAMLint_of_Cint"
external plus_cint : cint -> cint -> cint = "Cplus"
```

```
external print_cint : cint -> unit = "printCint" ;;
```

Nous pouvons maintenant utiliser des entiers sans « bit de tags », donc dans le format courant en C tout en conservant le GC d'Objective CAML. Par contre, ce ne sont plus des valeurs immédiates ce qui sera plus coûteux pour les opérations arithmétiques simples.

```
# let a = 1000000000 ;;  
val a : int = 1000000000  
# a+a ;;  
- : int = -147483648  
# let c = let b = cint_of_int a in plus_cint b b ;;  
val c : cint = <abstr>  
# print_cint c ; print_newline () ;;  
2000000000  
- : unit = ()  
# int_of_cint c ;;  
- : int = -147483648
```

Fonction de finalisation

Il est aussi possible de stocker dans un bloc abstrait des pointeurs vers des blocs alloués hors du tas Objective CAML. Nous savons qu'une valeur Objective CAML qui n'est plus utilisée par le programme est susceptible d'être récupérée par le GC. Mais que devient alors un bloc du tas C référencé par une valeur qui a disparue ? Pour éviter cette fuite de mémoire, nous pouvons associer une fonction dite de finalisation qui est exécutée par le GC avant de récupérer le bloc abstrait.

Pour ce faire nous utiliserons un bloc marqué par le tag `Final_tag`. Un tel bloc contient en plus des valeurs souhaitées, une fonction prenant comme argument le bloc lui-même et qui est exécutée avant la récupération de la valeur.

L'allocation de tels blocs se fait en utilisant `alloc_final (n, f, used, max)` :

- `n` est la taille du bloc. Attention, la première case du bloc est

utilisée pour stocker la fonction de finalisation.

- `f` est la fonction de finalisation : `void f (value);`
- `used` : place mémoire (hors du tas Objective CAML) supposée utilisée par la valeur.
- `max` : place mémoire maximale que l'on accepte de ne pas récupérer systématiquement.

Pour être plus efficace, le GC ne récupère pas un bloc dès qu'il devient inutilisé. Le rapport `used/max` permet de spécifier le nombre de blocs abstraits que le GC peut laisser alloués alors qu'ils ne sont plus utilisables.

Le programme suivant manipule des tableaux d'entiers C alloués dans le tas C par appel à `malloc`. Pour que ces tableaux puissent être bien récupérés lors d'un GC d'Objective CAML, la fonction `create` crée un bloc contenant le pointeur vers le tableau d'entiers (type `IntTab`) et la fonction de finalisation `finalize_it`.

```
#include <malloc.h>
#include <stdio.h>
#include <caml/mlvalues.h>

typedef struct {
    int size ;
    long * tab ; } IntTab ;

IntTab *alloc_it (int s)
{
    IntTab *res = malloc(sizeof(IntTab)) ;
    res->size = s ;
    res->tab = (long *) malloc(sizeof(long)*s) ;
    return res ;
}

void free_it (IntTab *p) { free(p->tab) ; free(p) ; }
void put_it (int n, long q, IntTab *p) { p->tab[n] = q ; }
long get_it (int n, IntTab *p) { return p->tab[n]; }

void finalize_it (value v)
{
    IntTab *p = (IntTab *) Field(v,1) ;
```

```

int i;
printf("recuperation d'un IntTab par finalisation [") ;
for (i=0;i<p->size;i++) printf("%d ",p->tab[i]) ;
printf("]\n"); fflush(stdout) ;
free_it ((IntTab *) Field(v,1)) ;
}
value create (value s)
{
  value bloc ;
  bloc = alloc_final (2, finalize_it,Int_val(s)*sizeof(IntTab),
  Field(bloc,1) = (value) alloc_it(Int_val(s)) ;
  return bloc ;
}
value put (value n,value q,value t)
{
  put_it (Int_val(n), Long_val(q), (IntTab *) Field(t,1)) ;
  return Val_unit ;
}
value get (value n,value t)
{
  long res = get_it (Int_val(n), (IntTab *) Field(t,1)) ;
  return Val_long(res) ;
}

```

Les fonctions visibles à partir d'Objective CAML seront : create, put et get.

```

# type c_int_array
external cia_create : int -> c_int_array = "create"
external cia_get : int -> c_int_array -> int = "get"
external cia_put : int-> int -> c_int_array -> unit = "put"

```

On peut désormais manipuler notre nouvelle structure de données :

```

# let tab = cia_create 10 and tab2 = cia_create 10
in for i=0 to 9 do cia_put i (i*2) tab done ;
for i=0 to 9 do print_int (cia_get i tab) ; print_string "\n" ;
print_newline () ;
for i=0 to 9 do cia_put (9-i) (cia_get i tab) tab2 done
for i=0 to 9 do print_int (cia_get i tab2) ; print_string "\n" ;

```

```
0 2 4 6 8 10 12 14 16 18
18 16 14 12 10 8 6 4 2 0 - : unit = ()
```

On force un GC pour provoquer l'appel à la fonction de finalisation :

```
# Gc.full_major () ;;
recuperation d'un IntTab par finalisation [18 16 14 12 10 8 6 4 2 0]
recuperation d'un IntTab par finalisation [0 2 4 6 8 10 12 14 16 18]
- : unit = ()
```

La fonction de finalisation peut aussi être utilisée pour fermer un fichier, terminer un processus, etc.

Paramètres, variables locales des fonctions C et GC

Il est possible d'invoquer le GC depuis une fonction C, les primitives pour le faire sont définies dans le fichier `memory.h`. La fonction `void Garbage_collection_function ()` est la primitive C d'invocation du GC. Selon que le programme est compilé en code-octet ou en code natif le GC mineur ou majeur est invoqué.

Examinons l'exemple suivant et voyons quelle erreur a pu se produire.

```
#include <caml/mlvalues.h>
#include <caml/memory.h>

value identite (value x)
{
  Garbage_collection_function() ;
  return x;
}

# external id : 'a -> 'a = "identite" ;;
external id : 'a -> 'a = "identite"
# id [1;2;3;4;5] ;;
- : int list = [539709628; 539709622; 539709616; 539709610; 539709604]
```

La liste passée en paramètre de id, donc de la fonction C identite, est susceptible d'être déplacée ou récupérée par le GC. Nous avons, dans notre exemple, forcé l'appel au GC, mais toute allocation d'un bloc aurait pu en faire tout autant. La liste, anonyme, passée en argument à id a été modifiée par le passage du GC car elle n'est pas connue dans l'ensemble des racines. Pour éviter celà, il faut utiliser une déclaration spécifiques des paramètres et des variables locales, qui sont des value, utilisés par les primitives C pour qu'ils soient correctement traités par le GC en utilisant les macro suivantes :

CAMLparam1(v)	:	pour une valeur,
CAMLparam2(v1,v2)	:	pour deux valeurs,
...		...
CAMLparam5(v1,...,v5)		pour cinq valeurs,
CAMLparam0 ;	:	obligatoire quand il n'y a pas de valeurs.

Si il y a plus de cinq paramètres de type value, les cinq premiers sont déclarés avec la macro CAMLparam5 et les suivants en utilisant le nombre de fois nécessaires les macros CAMLxparam1, ..., CAMLxparam5.

```
CAMLparam5(v1,...,v5);
CAMLxparam5(v6,...,v10);
CAMLxparam2(v11,v12);
```

pour douze valeurs

Les variables locales de type value doivent être elles-aussi déclarées auprès du GC en utilisant les macros CAMLlocal1, ..., CAMLlocal5. La déclaration d'un tableau de valeurs se fait avec la macro CAMLlocalN(tab,n) où n est la taille du tableau tab. Enfin, si la primitive retourne une value, nous utiliserons la macro CAMLreturn.

Voici donc la version correcte de l'exemple précédent :

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
value identite2 (value x)
{
    CAMLparam1(x) ;
    Garbage_collection_function() ;
```

```
CAMLreturn x;
}
```

```
# external id : 'a -> 'a = "identite2" ;;
external id : 'a -> 'a = "identite2"
# let a = id [1;2;3;4;5] ;;
val a : int list = [1; 2; 3; 4; 5]
```

Cette fois nous obtenons le résultat attendu.

Appel d'une fermeture Objective CAML depuis C

Pour, dans une fonction C, appliquer une fermeture (i.e. une valeur fonctionnelle Objective CAML) à un ou plusieurs arguments, nous utiliserons les macros définies dans le fichier `callback.h`.

<code>callback(f,v)</code>	:	une fermeture <code>f</code> appliquée à un argument <code>v</code> ,
<code>callback2(f,v1,v2)</code>	:	à deux arguments,
<code>callback3(f,v1,v2,v3)</code>	:	à trois arguments,
<code>callbackN(f,n,tab)</code>	:	à <code>n</code> arguments stockés dans le tableau <code>tab</code> .

Toutes ces macros retournent une valeur résultat de l'application.

Déclarations de fonctions Objective CAML

L'utilisation des macros `callback` nécessite d'avoir la fonction à appliquer sous la forme d'une fermeture, c'est-à-dire d'une valeur qu'il aura fallu passer en argument de la primitive. Il est aussi possible d'enregistrer une fermeture depuis un programme Objective CAML en lui associant un nom et de la récupérer dans un programme C grâce à son nom.

La fonction `register` du module `Callback` associe un nom (string) à une fermeture (de type quelconque donc `'a`). Cette fermeture peut-

être retrouvée depuis C grâce à la fonction C `caml_named_value` qui prend une chaîne de caractères et rend un pointeur vers la fermeture associée au nom, si elle existe, ou le pointeur nul, sinon.

Voyons cela sur un petit exemple :

```
# let plus x y = x + y ;;
val plus : int -> int -> int = <fun>
# Callback.register "plus3_ocaml" (plus 3);;
- : unit = ()
```

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/callback.h>
```

```
value plus3_C (value v)
{
  CAMLparam1(v);
  CAMLlocal1(f);
  f = * caml_named_value("plus3_ocaml") ;
  CAMLreturn callback(f,v) ;
}
```

```
# external plusC : int -> int = "plus3_C" ;;
external plusC : int -> int = "plus3_C"
# plusC 1 ;;
- : int = 4
# Callback.register "plus3_ocaml" (plus 5);;
- : unit = ()
# plusC 1 ;;
- : int = 6
```

Il n'y a pas exactement correspondance entre la déclaration d'une primitive C par **external** et l'enregistrement d'une fermeture Objective CAML par la fonction `register`. Dans le premier cas, il s'agit d'une déclaration statique, le lien entre les deux noms se fait à l'édition de lien. Alors que dans l'autre cas, il s'agit d'une liaison dynamique qui intervient durant l'exécution. En particulier, rien

n'interdit de remplacer la liaison nom--fermeture en remplaçant la fermeture et en modifiant ainsi le comportement des fonctions C utilisant ce nom.

Gestion des exceptions en C et en Objective CAML

Les mécanismes de rupture et reprise de calcul varient selon les langages : C utilise le couple `setjmp-longjmp` et les exceptions Objective CAML, la construction (**try .. with, raise**). Ces mécanismes ne sont, bien sûr, pas compatibles. C'est-à-dire qu'ils ne conservent pas les mêmes informations à la pose d'un récupérateur. Le fait de pouvoir imbriquer des récupérateurs d'exceptions de nature différente et d'en outrepasser certains pose de réelles difficultés d'implantation pour garantir la sûreté de leur utilisation. C'est pour cela que seules les exceptions Objective CAML sont traitées. Néanmoins rien n'empêche de les déclencher ou de les récupérer en C.

Toutes les macros et les fonctions utilisées dans cette partie sont définies dans le fichier `fail.h`.

Déclencher une exception prédéfinie

Il est facilement possible de déclencher, depuis une fonction C, les exceptions `Failure` ou `Invalid_argument` du module `Pervasives` en utilisant les fonctions suivantes :

<code>failwith(s)</code>	:	déclenche l'exception <i>Failure(s)</i>
<code>invalid_argument(s)</code>	:	déclenche l'exception <i>Invalid_argument(s)</i> .

Dans les deux cas, `s` est une chaîne de caractères C (`char *`).

Déclencher une exception quelconque

Le mécanisme pour lever une exception quelconque définie en

Objective CAML est semblable à celui utilisé pour appliquer une fermeture. Il faut dans un premier temps enregistrer l'exception par la fonction `register_exception` du module `Callback`. On récupère ensuite la valeur avec la fonction `caml_named_value` (voir page ??). On déclenche l'exception avec l'une des fonctions suivantes :

<code>raise_constant(e)</code>	lève l'exception <code>e</code> ,
<code>raise_with_arg(e,v)</code>	lève l'exception <code>e</code> avec <code>v</code> pour argument,
<code>raise_with_string(e,s)</code>	idem, mais la valeur est une copie de la chaîne <code>s</code> .

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/fail.h>
```

```
value division (value v1,value v2)
{
  CAMLparam2 (v1,v2);
  if (!Long_val(v2))
    raise_with_arg(*caml_named_value("exception"),v1) ;
  CAMLreturn Val_long(Long_val(v1)/Long_val(v2)) ;
}
```

```
# external divide : int -> int -> int = "division" ;;
external divide : int -> int -> int = "division"
# exception Division_par_zero of int ;;
exception Division_par_zero of int
# Callback.register_exception "exception" (Division_par_
- : unit = ()
# divide 20 4 ;;
- : int = 5
# divide 22 0 ;;
Uncaught exception: Division_par_zero(22)
```

Rattraper une exception

Il est possible de rattraper une exception depuis une fonction C, mais seulement si elle a été déclenchée par l'application d'une fermeture venue d'Objective CAML et non directement depuis une fonction C. Les macros `callback_exn`, `callback2_exn`, `callback3_exn` et `callbackN_exn` fonctionnent comme les macros `callback` simples, sauf que si l'application a déclenché une exception, celle-ci est rendue comme résultat. Le résultat d'un `callback_exn` peut être testé par le prédicat `Is_exception_result(v)` qui retourne 1 si la valeur provient du déclenchement d'une exception et 0 sinon. On retrouve le constructeur de l'exception par appel à la fonction `Extract_exception(v)`.

La fonction `C` `divise_affiche` appelle une fonction `divise` et teste si le retour est une exception, en utilisant l'appel `callback2_exn`. Si c'est le cas, elle affiche une trace et la redéclenche, sinon elle affiche le résultat.

```
#include <stdio.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/callback.h>
#include <caml/fail.h>

value divise_affiche (value v1,value v2)
{
  CAMLparam2 (v1,v2) ;
  CAMLlocal3 (div,dbz,res) ;
  div = * caml_named_value("divise") ;
  dbz = * caml_named_value("div_by_0") ;
  res = callback2_exn (div,v1,v2) ;
  if (Is_exception_result(res))
  {
    value exn=Extract_exception(res);
    if (Field(exn,0)==dbz) printf("division par 0\n") ;
    else printf("autre exception\n");
    fflush(stdout);
    if (Wosize_val(exn)==1) raise_constant(Field(exn,0)) ;
    else raise_with_arg(Field(exn,0),Field(exn,1)) ;
  }
  printf("resultat = %d\n",Long_val(res)) ;
}
```

```

fflush(stdout) ;
CAMLreturn Val_unit ;
}

# Callback.register "divise" (/) ;;
- : unit = ()
# Callback.register_exception "div_by_0" Division_by_zero
- : unit = ()
# external divise_ml : int -> int -> unit = "divise_affi
external divise_ml : int -> int -> unit = "divise_affich
# divise_ml 42 3 ;;
resultat = 14
- : unit = ()
# divise_ml 21 0 ;;
division par 0
Uncaught exception: Division_by_zero

```

Il est donc possible de déclencher une exception en C et de la rattraper en Objective CAML; de même, il est possible de déclencher une exception depuis Objective CAML et la rattraper en C. En revanche, un programme C ne peut pas à lui seul déclencher et rattraper une exception d'Objective CAML.

Programme principal en C

Jusqu'à maintenant le point d'entrée du programme était en Objective CAML. Rien n'empêche de le faire démarrer en C. Pour cela il faudra définir, dans le programme C, la fonction `main` habituelle. Elle initialisera le runtime d'Objective CAML par la fonction `caml_main(char **)` qui prend en argument le tableau d'arguments correspondant à la valeur du tableau `Sys.argv` d'Objective CAML. Le contrôle peut être passé à ce dernier en utilisant un callback (voir page [??](#)).

Inclure du code Objective CAML en C

Le compilateur Objective CAML permet d'engendrer des fichiers

objets C (fichiers avec l'extension '.o') en lieu et place des fichiers objets Objective CAML (d'extension '.cmo' ou '.cmx'). Il suffit d'utiliser l'option `-output-obj` du compilateur et d'ajouter la bibliothèque `unix.cma`.

```
ocamlc -output-obj unix.cma fichiers.ml
ocamlopt -output-obj unix.cmxa fichiers.ml
```

À partir des fichiers sources en Objective CAML `fichiers.ml`, le fichier objet portant par défaut le nom `camlprog.o` est engendré.

Un exécutable est obtenu en effectuant l'édition de liens avec un compilateur C en ajoutant la bibliothèque `libcamlrun.a` pour un programme compilé en code-octet ou la bibliothèque `libasmrun.a` pour la compilation en code natif.

```
cc camlprog.o fichiersC.o -lcamlrun
cc camlprog.o fichiersC.o -lasmrn
```

L'invocation d'une fonction Objective CAML depuis le programme C se fait comme décrit précédemment par les fonctions callback. La seule différence est que l'initialisation et le lancement du runtime d'Objective CAML se fait par la fonction `caml_startup` à la place de `caml_main`.

Exercices

Affichage polymorphe

On cherche à définir une fonction d'affichage `print` de type `'a -> unit` pouvant afficher n'importe quelle valeur Objective CAML. Pour cela on reprend la fonction `inspecte` pour l'améliorer.

1. Écrire en C la **fonction** `print_ws` qui affiche les valeurs Objective CAML de la manière suivante :

- valeurs immédiates : comme des entiers C
- chaînes : entre `""`

- flottants : habituels
- tableau de flottants : entre [| |]
- les fermetures : < code, env >
- tout le reste comme un n-uplet, entre ()

La fonction parcourt les valeurs de manière récursive.

```
#include <stdio.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>

value print_ws (value v) {
    CAMLparam1(v);
    int taille,i ;
    if (Is_long(v)) printf("%d", Long_val(v));
    else {
        taille=Wosize_val(v);
        switch (Tag_val(v))
        {
            case String_tag :
                printf("\'%s\'", String_val(v));
                break;
            case Double_tag:
                printf("%g", Double_val(v));
                break;
            case Double_array_tag :
                printf (" [| |");
                if (taille>0) printf("%g", Double_field(v,0)
                for (i=1;i<(taille/2);i++) printf("; %g", Double_
                printf(" |");
                break;
            case Abstract_tag :
            case Final_tag :
                printf("<abstract>");
                break;
            case Closure_tag :
                printf("<%d, ",Code_val(v)) ;
                if (taille>1) print_ws(Field(v,1)) ;
                for (i=2;i<taille;i++) {
```

```

        printf("; ") ;
        print_ws(Field(v,i));
    }
    printf(">");
    break;
    default:
    if (Tag_val(v)>=No_scan_tag) printf("?");
    else {
        printf("(");
        if (taille>0) print_ws(Field(v,0));
        for (i=1;i<taille;i++) {
            printf(", ");
            print_ws(Field(v,i));
        }
        printf(")");
    }
    }
    }
    fflush(stdout);
    return Val_unit;
}

```

8. Pour éviter de boucler sur une valeur circulaire et pour montrer le partage, modifier cette fonction pour tenir compte des adresses déjà rencontrées. Si une [adresse](#) apparaît plusieurs fois, alors lui donner un nom au premier affichage (`v = nom`) et utiliser ce nom par la suite quand vous la rencontrez de nouveau.

1. Définir une [structure](#) de données pour [stocker](#) les adresses, [indiquer](#) quand elles occurrent plusieurs fois et leur [associer](#) un nom.

```

# let (gensym,init_gensym) =
  let i = ref 0 in
    (function () -> incr i ; "val_" ^ (string_of_int i)) ,
    (function () -> i:=0) ;;
val gensym : unit -> string = <fun>
val init_gensym : unit -> unit = <fun>

```

```

# type occurrence =
  Une_fois
  | Plusieurs_fois
  | Deja_nomme of string ;;
type occurrence = | Une_fois | Plusieurs_fois |

# let table = Hashtbl.create 17 ;;
val table : ('_a, '_b) Hashtbl.t = <abstr>

# let ajoute (adr : adresse) =
  try
    match Hashtbl.find table adr with
    Une_fois -> Hashtbl.remove table adr ;
    Hashtbl.add table adr Plusieurs_fois ;
    true
  | _ -> true
  with
    Not_found -> Hashtbl.add table adr Une_fois ;
val ajoute : adresse -> bool = <fun>

# let multiple_occ adr =
  match Hashtbl.find table adr with
  Une_fois -> false
  | _ -> true ;;
val multiple_occ : adresse -> bool = <fun>

# let deja_nomme adr =
  match Hashtbl.find table adr with
  Une_fois -> failwith "deja_nomme"
  | Plusieurs_fois -> Hashtbl.remove table adr ;
  Hashtbl.add table adr (Deja_nomme (gensym ()))
  false
  | _ -> true ;;
val deja_nomme : adresse -> bool = <fun>

# let nom_de adr =
  match Hashtbl.find table adr with

```

```

    Deja_nomme s -> s
  | _ -> raise Not_found ;;
val nom_de : adresse -> string = <fun>

```

2. La valeur est parcourue une première fois pour [explorer](#) toutes les adresses et les enregistrer dans la table.

```

# Callback.register "add" ajoute ;;
- : unit = ()
# Callback.register "multiple?" multiple_occ ;;
- : unit = ()
# Callback.register "named?" deja_nomme ;;
- : unit = ()
# Callback.register "name" nom_de ;;
- : unit = ()

# external explore_value : 'a -> unit = "explore"
external explore_value : 'a -> unit = "explore"

```

3. Le second parcours réalise l'[affichage](#) proprement dit en nommant les adresses à la première de leurs occurrences.

```

# external print_rec : 'a -> unit = "print_gen"
external print_rec : 'a -> unit = "print_gen"

```

4. Définir la fonction [print](#) réunissant le tout.

```

external print_rec : 'a -> unit = "print_gen" ;
let print v =
  Hashtbl.clear table ;
  init_gensym () ;
  explore_value v ;
  print_rec v ;;

```

```

# type adresse ;;
type adresse

```

Produit de matrices : partage ou copie

1. Définir un **type abstrait** *float_matrix* pour des matrices de flottants.

```
# type float_matrix ;;  
type float_matrix
```

2. Définir un **type C** pour de telles matrices.

```
typedef struct { int size_x , size_y ;  
                float * mat ;  
                } Matrix ;
```

3. Écrire une fonction en C de **conversion** de valeur de type *float array array* vers une valeur de type *float_matrix*.

```
value conversion_to_C (value faa) {  
  CAMLparam1(faa) ;  
  CAMLlocal1(vres) ;  
  Matrix * res ;  
  int size_x, size_y ;  
  
  /* taille du vecteur de vecteur */  
  size_x = Wosize_val(faa) ;  
  if (size_x>0) size_y = Wosize_val(Field(faa,0))/2  
  
  /* allocation de la valeur float_matrix */  
  vres=alloc(sizeof(Matrix),Abstract_tag) ;  
  res=(Matrix *) vres ;  
  res->size_x = size_x ;  
  res->size_y = size_y ;
```



```

if (size_x*size_y==0) res->mat=0 ;
else {
    int i,j ;
    float * tab ;
    value vect ;
    res->mat=tab=(float *) alloc(sizeof(float)*size_x*size_y) ;
    for (i=0;i<size_x;i++) {
        vect = Field(faa,i) ;
        for (j=0;j<size_y;j++) *(tab++) = Double_field(faa,vect,j) ;
    }
}
CAMLreturn vres ;
}

```

4. Écrire la fonction en C effectuant la conversion [inverse](#).

```

value conversion_to_Caml (value matrix) {
    CAMLparam1(matrix) ;
    CAMLlocal2(res,aux) ;
    Matrix* mat = (Matrix *) matrix ;
    float * tab = mat->mat ;
    int size = mat->size_x*mat->size_y ;
    int i,j ;
    res=alloc(mat->size_x,0);
    for (i=0;i<mat->size_x;i++) {
        aux = alloc(2*mat->size_y,Double_array_tag) ;
        Field(res,i) = aux ;
        for (j=0;j<mat->size_y;j++) Store_double_field(aux,j,tab[i*mat->size_y+j]) ;
    }
    CAMLreturn res ;
}

```

5. Ajouter en C les fonctions pour la [somme](#) et le [produit](#) de telles matrices.

```

value plus (value arg1,value arg2) {
    CAMLparam2(arg1,arg2) ;
    CAMLlocal1(vres) ;
    Matrix *m1=(Matrix*) arg1, *m2=(Matrix*) arg2,*re

```

```

float *tab;
int i,size=m1->size_x*m1->size_y ;
vres=alloc(sizeof(Matrix),Abstract_tag) ;
res =(Matrix*) vres;
res->size_x=m1->size_x;
res->size_y=m1->size_y;
res->mat=tab=(float *) alloc(sizeof(float)*size,A
for (i=0;i<size;i++) tab[i]=m1->mat[i]+m2->mat[i]
CAMLreturn vres;
}

value prod (value arg1,value arg2) {
  CAMLparam2(arg1,arg2) ;
  CAMLlocal1(vres) ;
  Matrix *m1=(Matrix*) arg1, *m2=(Matrix*) arg2,*re
  float *tab;
  int i,j,k;
  vres=alloc(sizeof(Matrix),Abstract_tag) ;
  res =(Matrix*) vres;
  res->size_x=m1->size_x;
  res->size_y=m2->size_y;
  res->mat=tab=(float *) alloc(sizeof(float)*res->s
                                Abstract_tag) ;
  for (i=0;i<res->size_x;i++)
    for (j=0;j<res->size_y;j++) {
      float acc=0 ;
      for (k=0;k<m1->size_y;k++)
        acc += m1->mat[i*m1->size_x+k] * m1->mat[k*m2->siz
        tab[i*m1->size_x+j]=acc ;
    }
  CAMLreturn vres;
}

```

6. Les [interfacer](#) avec Objective CAML et les utiliser.

```

# external to_matrix : float array array -> float_m
# external of_matrix : float_matrix -> float array
# external somme : float_matrix -> float_matrix ->
# external produit : float_matrix -> float_matrix -

```

Compter les mots : programme principal en C

La commande Unix `wc` compte le nombre de caractères, de mots et de lignes d'un fichier. Cet exercice a pour but de réaliser cette commande en ne comptant qu'une fois les mots qui sont répétés plusieurs fois.

1. Écrire le programme `wc` en C comptant simplement les mots, les lignes et les caractères d'un fichier dont le nom est passé en argument.

```
#include <stdio.h>

void read_file (char *path) {
    FILE *fd=fopen(path,"r");
    int car=0;
    char buffer[80], *buff;
    int nb_car=0, nb_mots=0, nb_lignes=0;

    buff=buffer; *buff=0;
    if (!fd) exit(-1) ;
    while ((car=getc(fd))!=EOF) {
        nb_car++ ;
        if (car=='\n') nb_lignes++ ;
        if (car==' ' || car=='\n' || (buff-buffer)>=80)
            if (buff!=buffer) { nb_mots++; buff=0; buff=b
        }
        else *(buff++)=car;
    }
    printf(" %d - %d - %d : %s\n",nb_lignes,nb_mots,nb_car);
}

int main (int argc,char **argv)
{
    if (argc>1) read_file(argv[1]);
    return 0;
}
```

```
}
```

2. Écrire en Objective CAML une fonction `ajoute_mot` qui à l'aide d'une table de hachage conserve le nombre de fois où la fonction a été invoquée avec pour argument une même chaîne de caractères.

```
# let table = Hashtbl.create 17 ;;  
val table : ('_a, '_b) Hashtbl.t = <abstr>  
  
# let ajoute_mot (m:string) =  
  try let p = Hashtbl.find table m in incr p  
  with Not_found -> Hashtbl.add table m (ref 1) ;;  
val ajoute_mot : string -> unit = <fun>
```

3. Écrire deux fonctions `nb_mots_repetes` et `nb_mots_differeents` comptant respectivement les mots répétés et les mots différents à partir de la table construite par `ajoute_mot`.

```
# let nb_mots_repetes () =  
  let i = ref 0 in  
  Hashtbl.iter (fun _ n -> if !n>1 then incr i) table  
  !i ;;  
val nb_mots_repetes : unit -> int = <fun>  
  
# let nb_mots_differeents () =  
  let i = ref 0 in  
  Hashtbl.iter (fun _ _ -> incr i) table ;  
  !i ;;  
val nb_mots_differeents : unit -> int = <fun>
```

4. Enregistrer les trois fonctions précédentes de sorte qu'elles puissent être utilisées depuis un programme C.

```
# Callback.register "add word" ajoute_mot ;  
  Callback.register "rep words" nb_mots_repetes ;
```

```
    Callback.register "diff words" nb_mots_differe
- : unit = ()
```

5. Réécrire la fonction **principale** du programme `wc` pour que soit affiché le nombre de mots différents en lieu et place du nombre de mots.

```
#include <caml/mlvalues.h>
#include <caml/callback.h>
```

6. Donner la fonction **main** et les commandes nécessaires pour compiler ce programme comme un programme Objective CAML.

```
int main (int argc, char **argv)
{
    caml_main(argv);
    if (argc>1) read_file(argv[1]);
    return 0;
}
```

7. Donner la fonction **main** et les **commandes** nécessaires pour compiler ce programme comme un programme C.

```
int main (int argc, char **argv)
{
    caml_startup(argv);
    if (argc>1) read_file(argv[1]);
    return 0;
}
```

Résumé

Ce chapitre a présenté l'interface entre le langage Objective CAML et le langage C. Celle-ci fournit les outils pour manipuler les valeurs Objective CAML en C. De plus les types abstraits d'Objective CAML permettent la manipulation inverse. Son principal avantage est de

permettre d'utiliser le mécanisme de GC d'Objective CAML pour des valeurs créées en C, langage dépourvu d'une telle récupération automatique de mémoire. Cela permet de composer une application à partir de composants développés dans les deux langages. Enfin les exceptions Objective CAML peuvent être manipulées et déclenchées en C. L'inverse n'est pas possible avec ces outils.

Pour en savoir plus

Pour bien comprendre le passage d'arguments en C et la représentation des données, l'ouvrage *Langage C : manuel de référence* ([[HS90](#)]) répond aux différentes interrogations.

Pour la partie extension du langage C, c'est-à-dire sur les exceptions et le GC, plusieurs travaux ajoutent ces fonctionnalités dont C est dépourvu. Le lecteur intéressé par les exceptions en C pourra lire le rapport technique ([[Rob89](#)]) qui décrit comment implanter un tel mécanisme en utilisant des macros non fermées et les ruptures de calcul `setjmp/longjmp` de C. La page de Hans Boehm contient une distribution d'une bibliothèque pour un GC conservatif à racines ambiguës pour C :

Lien: http://reality.sgi.com/boehm_mti/

Pour la partie interopérabilité entre Objective CAML et C, les outils décrits dans ce chapitre restent de bas niveau et assez fastidieux d'emploi. Néanmoins ils permettent de préciser la copie ou le partage entre données des deux mondes. Il existe un outil de plus haut niveau, appelé CamlIDL qui permet de créer automatiquement les « souches » (en anglais stub), ou fonctions d'encapsulation, pour l'appel de fonctions C et la conversion entre types de données. Les types et fonctions C sont décrits dans un langage IDL (pour Interface Definition Language), proche de C++ donc incluant C. Cette description est ensuite passée à CamlIDL pour produire les fichiers `.mli`, `.ml` et `.c` correspondant à cette encapsulation. La distribution de cet outil s'obtient à la page suivante :

Lien: <http://caml.inria.fr/camlidl/>

Il existe d'autres interfaces entre Objective CAML et d'autres

langages. On trouvera ces contributions à la page de la « bosse du chameau » :

Lien: <http://caml.inria.fr/hump.html>

On peut citer plusieurs essais avec le langage Fortran ainsi qu'avec un interprète de byte-code Objective CAML écrit en Java.

Enfin l'interopérabilité entre Objective CAML et d'autres langages peut aussi être réalisée par communication de valeurs sur un réseau. Ce thème sera abordé au chapitre sur la programmation répartie (20).

Applications

Introduction

Les applications de ce chapitre cherchent à illustrer les nombreux concepts de programmation présentés dans cette partie.

La première application est la construction d'une bibliothèque de composants graphiques, appelée Upi (Une Petite Interface). On l'appliquera à la réalisation d'un convertisseur Francs/Euros. Les composants décrits ici réagissent aux interactions d'un utilisateur en déclenchant des fonctions de traitement. Bien que la difficulté algorithmique de cette application reste simple, son intérêt est de montrer la communication entre composants grâce aux fermetures. En effet les différentes fonctions de traitement partagent certaines valeurs dans leur environnement. Il est nécessaire de connaître la bibliothèque de base Graphics pour apprécier la construction de la bibliothèque Upi (voir chapitre 5, page ??).

La deuxième application est une recherche du chemin de moindre coût dans un graphe orienté. Elle suit l'algorithme de Dijkstra qui calcule tous les chemins de moindre coût d'un sommet source vers tous les autres sommets atteignables à partir de cette source. Pour accélérer les recherches demandées, on implante un mécanisme de cache qui conserve les recherches déjà effectuées en utilisant un tableau de pointeurs faibles (voir page ??). Le GC peut libérer des

éléments de ce tableau, mais ils sont toujours recalculables en cas de besoin. La visualisation d'un graphe utilise les boutons simples de la bibliothèque Upi pour la sélection des sommets origine et destination du parcours demandé. On compare l'efficacité d'exécution entre les versions avec et sans cache. Pour faciliter les mesures de temps de ces différentes versions, on passe en argument de l'application le fichier de description du graphe ainsi que l'origine et la destination du parcours recherché. On dotera le programme de recherche d'< < une petite interface graphique > >.

Construction d'une interface graphique

La réalisation de l'interface graphique d'un programme est un travail fastidieux si les outils pour le faire ne sont pas assez riches. C'est le cas avec la bibliothèque Graphics. Le confort d'utilisation d'un programme provient en partie de son interface. Pour cela nous allons tout d'abord construire, au dessus de Graphics, une nouvelle bibliothèque, appelée Upi et l'utiliser en tant que module simple pour la construction de l'interface d'une application.

Cette interface graphique manipule des composants. Un composant est une zone de la fenêtre principale qui peut être affichée, dans un certain contexte graphique, et traiter des événements qui lui sont destinés. Il y a principalement deux types de composants : les composants simples, comme un bouton de confirmation ou une zone de saisie d'un texte, et les conteneurs qui acceptent de recevoir dans leur zone d'autres composants. Un composant ne peut être attaché qu'à un seul conteneur. L'interface d'une application devient alors un arbre dont la racine correspond au conteneur principal (la fenêtre graphique), les noeuds sont d'autres conteneurs et les feuilles des composants simples ou des conteneurs sans descendant. Cette arborescence facilite la propagation des événements produits par l'interaction avec l'utilisateur. Si un composant reçoit un événement, il vérifie si l'un de ses descendants peut le traiter, si oui il le lui envoie et sinon il effectue l'action associée à cet événement.

Le composant est l'élément essentiel de cette bibliothèque. Nous le définissons comme un enregistrement qui contient principalement des informations de taille, un contexte graphique, le composant père et les composants fils, ainsi que les fonctions d'affichage et de

traitement des événements. Les conteneurs possèdent une fonction de placement des composants. Pour pouvoir définir le type *component*, on construit les types pour le contexte graphique, les événements et les options de création. Un contexte graphique est utilisé pour les informations de « styles graphiques » comme les couleurs de fond et de tracé, la largeur des traits, le point courant du composant et les polices de caractères choisies. On définit ensuite la nature des événements envoyés aux composants. Ceux-ci sont plus variés que ceux de la bibliothèque Graphics sur lesquels ils reposent. On intègre un mécanisme simple d'options qui facilite la spécialisation de contextes graphiques ou de composants. Une difficulté d'implantation provient du placement d'un composant dans un conteneur.

La boucle générale de gestion des événements reçoit des événements physiques par la fonction d'écoute de la bibliothèque Graphics, détermine si d'autres événements sont conséquence de ces événements physiques puis les envoie au composant racine. On décrit ensuite les composants suivants : textes à afficher, boutons, listes de choix, zones de saisie et composants enrichis.

On montre l'assemblage de composants pour la construction d'interfaces graphiques, en l'illustrant par un convertisseur Francs/Euros. Les divers composants de cette application communiquent entre eux en partageant un état.

Contexte graphique, événements et options

On définit les types de base, et les fonctions de création et de modification, des contextes graphiques, des événements et des options. Ce dernier type facilite la paramétrisation des fonctions de création d'objets graphiques.

Contexte graphique

Le contexte graphique permet de tenir compte des couleurs de fond et d'encre, de la police de caractère, de la largeur des traits et du point courant. Le type suivant découle de cette définition.

```
# type g_context = {
  mutable bcol : Graphics.color;
  mutable fcol : Graphics.color;
  mutable font : string;
  mutable font_size : int;
  mutable lw : int;
  mutable x : int;
  mutable y : int };;
```

La fonction `make_default_context` crée un nouveau contexte graphique contenant des valeurs par défaut¹.

```
# let default_font = "fixed"
let default_font_size = 12
let make_default_context () =
{ bcol = Graphics.white; fcol = Graphics.black;
  font = default_font;
  font_size = default_font_size;
  lw = 1;
  x = 0; y = 0};;;
val default_font : string = "fixed"
val default_font_size : int = 12
val make_default_context : unit -> g_context = <fun>
```

Les fonctions d'accès aux champs permettent de récupérer les différentes valeurs d'un contexte graphique sans connaître la représentation du type.

```
# let get_gc_bcol gc = gc.bcol
let get_gc_fcol gc = gc.fcol
let get_gc_font gc = gc.font
let get_gc_font_size gc = gc.font_size
let get_gc_lw gc = gc.lw
let get_gc_cur gc = (gc.x,gc.y);;
val get_gc_bcol : g_context -> Graphics.color = <fun>
val get_gc_fcol : g_context -> Graphics.color = <fun>
val get_gc_font : g_context -> string = <fun>
val get_gc_font_size : g_context -> int = <fun>
val get_gc_lw : g_context -> int = <fun>
val get_gc_cur : g_context -> int * int = <fun>
```

Les fonctions de modification des champs reposent sur le même modèle.

```
# let set_gc_bcol gc c = gc.bcol <- c
let set_gc_fcol gc c = gc.fcol <- c
let set_gc_font gc f = gc.font <- f
let set_gc_font_size gc s = gc.font_size <- s
let set_gc_lw gc i = gc.lw <- i
let set_gc_cur gc (a,b) = gc.x<- a; gc.y<-b;;
val set_gc_bcol : g_context -> Graphics.color -> unit =
val set_gc_fcol : g_context -> Graphics.color -> unit =
val set_gc_font : g_context -> string -> unit = <fun>
val set_gc_font_size : g_context -> int -> unit = <fun>
val set_gc_lw : g_context -> int -> unit = <fun>
val set_gc_cur : g_context -> int * int -> unit = <fun>
```

On peut ainsi créer de nouveaux contextes, avoir accès aux différents champs d'un contexte et modifier un des champs d'une valeur de type *g_context*.

La fonction *use_gc* applique à la fenêtre graphique les informations du contexte graphique.

```
# let use_gc gc =
  Graphics.set_color (get_gc_fcol gc);
  Graphics.set_font (get_gc_font gc);
  Graphics.set_text_size (get_gc_font_size gc);
  Graphics.set_line_width (get_gc_lw gc);
  let (a,b) = get_gc_cur gc in Graphics.moveto a b;;
val use_gc : g_context -> unit = <fun>
```

Certaines informations, comme la couleur du fond, ne sont pas directement exploitables par la bibliothèque Graphics et n'apparaissent pas dans la fonction *use_gc*.

Événements

La bibliothèque Graphics ne contient qu'un nombre limité d'événements d'interaction : clic souris, déplacement de souris et

appui de touche. On désire enrichir la nature des événements survenant dans l'interaction en intégrant des événements propres aux composants. Pour cela on définit le type *rich_event* :

```
# type rich_event =  
  MouseDown | MouseUp | MouseDrag | MouseMove  
  | MouseEnter | MouseExit | Exposure  
  | GotFocus | LostFocus | KeyPress | KeyRelease;;
```

Pour pouvoir engendrer de tels événements, il est nécessaire de conserver un historique des derniers événements produits. Les événements *MouseDown* et *MouseMove* correspondent respectivement aux événements de la souris (clic et déplacement) gérés par *Graphics*. Les autres événements souris sont créés en fonction soit de l'avant dernier événement (*MouseUp*) ou du dernier composant ayant traité un événement physique (*MouseExit*). L'événement *Exposure* correspond à une demande de réaffichage du composant. La notion de focus exprime l'attention particulière d'un composant à un type d'événement. Typiquement, la saisie d'un texte dans un composant ayant accaparé le focus du clavier indique qu'il est le seul à traiter les événements *KeyPress* et *KeyRelease*. Un événement *MouseDown* sur un composant `< >` lui donnera le focus clavier en le retirant du composant qui le possédait éventuellement.

La création de ces nouveaux événements est contrôlée par la boucle de gestion des événements, décrite page ??.

Options

Une interface graphique a besoin d'une règle pour le passage et le décodage des options de création des objets graphiques (composants, contextes graphiques). Si l'on veut créer un contexte graphique avec des couleurs spécifiques, il est actuellement nécessaire de construire un contexte graphique avec les valeurs par défaut puis d'appeler les deux fonctions de modification des champs de couleur sur ce contexte. Dans le cas d'objets graphiques plus complexes, cela peut devenir très vite fastidieux. Ces options doivent pouvoir être étendues ce qui permet d'augmenter les composants de la bibliothèque. Il y a besoin d'un type somme « extensible ». Le seul fourni par Objective CAML est le type *exn*

pour les exceptions. Comme il est préférable pour la clarté des programmes de n'utiliser ce type que pour de vraies exceptions, on simule un type somme extensible par des faux constructeurs représentés par des chaînes de caractères. On définit le type *opt_val* pour les valeurs des options. Une option devient un couple dont le premier élément est le nom de l'option et le deuxième sa valeur. Le type *lopt* regroupe une liste d'options.

```
# type opt_val = Copt of Graphics.color | Sopt of string
| Iopt of int | Bopt of bool;;
# type lopt = (string * opt_val) list ;;
```

Les fonctions de décodage prennent en entrée une liste d'options, un nom d'option et une valeur par défaut. Si le nom appartient à la liste alors la valeur associée est retournée, sinon c'est la valeur par défaut. On ne décrit que les fonctions de décodage des options entières et booléennes, les autres reposent sur le même modèle.

```
# exception OptErr;;
exception OptErr
# let theInt lo name default =
  try
  match List.assoc name lo with
  | Iopt i -> i
  | _ -> raise OptErr
  with Not_found -> default;;
val theInt : ('a * opt_val) list -> 'a -> int -> int = <
# let theBool lo name default =
  try
  match List.assoc name lo with
  | Bopt b -> b
  | _ -> raise OptErr
  with Not_found -> default;;
val theBool : ('a * opt_val) list -> 'a -> bool -> bool
```

On peut donc écrire une fonction de mise à jour d'un contexte graphique en passant une liste d'options de la manière suivante :

```
# let set_gc gc lopt =
  set_gc_bcol gc (theColor lopt "Background" (get_gc_bcol
  set_gc_fcol gc (theColor lopt "Foreground" (get_gc_fcol
```

```

set_gc_font gc (theString lopt "Font" (get_gc_font gc))
set_gc_font_size gc (theInt lopt "FontSize" (get_gc_font_size gc))
set_gc_lw gc (theInt lopt "LineWidth" (get_gc_lw gc));;
val set_gc : g_context -> (string * opt_val) list -> unit

```

Cela permet de ne pas tenir compte de l'ordre des options passées.

```

# let dc = make_default_context () in
  set_gc dc [ "Foreground", Copt Graphics.blue;
    "Background", Copt Graphics.yellow];
  dc;;
- : g_context =
{bcol=16776960; fcol=255; font="fixed"; font_size=12; lw=1;

```

On obtient un système assez souple qui malheureusement échappe en partie au système de types. Le nom d'une option est du type *string* et rien n'empêche de construire un nom inexistant. La conséquence est alors de ne pas tenir compte de sa valeur.

Composants et conteneurs

Le composant est la brique essentielle de cette bibliothèque. On désire avoir la possibilité de créer des composants et de facilement les assembler pour construire des interfaces. Ils doivent pouvoir s'afficher, savoir qu'un événement leur est destiné et traiter cet événement. Quand ils sont conteneurs, ils doivent pouvoir recevoir d'autres composants ou en retirer. On suppose qu'un composant simple ne peut être ajouté qu'à un seul conteneur.

Création de composants

Une valeur de type *component* possède une taille (w et h), une position absolue dans la fenêtre principale (x et y), un contexte graphique pour l'affichage (gc), un indicateur pour savoir s'il est un conteneur (container), un père s'il est attaché à un conteneur (father), une liste de ses fils (sons) et quatre fonctions permettant le placement d'autres composants (layout), son affichage (display), l'appartenance d'un point à sa zone (mem) et une fonction de traitement d'un événement (listener) qui retourne true si

l'événement est traité et false sinon. Le paramètre de listener prend un état enrichi (type *rich_status*) qui contient le nom de l'événement, l'état de l'événement physique provenant de Graphics, les informations sur le focus clavier et le focus général, ainsi que le dernier composant ayant traité un événement. On obtient les déclarations mutuellement récursives suivantes :

```
# type component =
{ mutable info : string;
  mutable x : int; mutable y : int;
  mutable w : int ; mutable h : int;
  mutable gc : g_context;
  mutable container : bool;
  mutable father : component list;
  mutable sons : component list;
  mutable layout_options : lopt;
  mutable layout : component -> lopt -> unit;
  mutable display : unit -> unit;
  mutable mem : int * int -> bool;
  mutable listener : rich_status -> bool }
and rich_status =
{ re : rich_event;
  stat : Graphics.status;
  mutable key_focus : component;
  mutable gen_focus : component;
  mutable last : component};;
```

On accède aux champs informatifs d'un composant par les fonctions suivantes.

```
# let get_gc c = c.gc;;
val get_gc : component -> g_context = <fun>
# let is_container c = c.container;;
val is_container : component -> bool = <fun>
```

Les trois fonctions suivantes vont définir le comportement par défaut d'un composant. La fonction d'appartenance (*in_rect*) vérifie que la coordonnée passée est bien dans le rectangle défini par les coordonnées du composant. La fonction d'affichage par défaut (*display_rect*) remplit le rectangle du composant par la couleur de fond de son contexte graphique. Le placement par défaut

(direct_layout) est celui du positionnement relatif du composant dans le conteneur. Les options acceptées sont "PosX" et "PosY" correspondant aux positions relatives du composant dans le conteneur.

```
# let in_rect c (xp,yp) =
  (xp >= c.x) && (xp < c.x + c.w) && (yp >= c.y) && (yp < c.y + c.h)
val in_rect : component -> int * int -> bool = <fun>
# let display_rect c () =
  let gc = get_gc c in
  Graphics.set_color (get_gc_bcol gc);
  Graphics.fill_rect c.x c.y c.w c.h ;;
val display_rect : component -> unit -> unit = <fun>
# let direct_layout c cl lopt =
  let px = theInt lopt "PosX" 0
  and py = theInt lopt "PosY" 0 in
  cl.x <- c.x + px; cl.y <- c.y + py ;;
val direct_layout :
  component -> component -> (string * opt_val) list -> unit
```

Il est maintenant possible de définir un composant grâce à la fonction create_component qui prend une largeur et une hauteur et qui utilise les trois fonctions précédentes.

```
# let create_component iw ih =
  let dc =
    {info="Anonymous";
     x=0; y=0; w=iw; h=ih;
     gc = make_default_context();
     container = false;
     father = []; sons = [];
     layout_options = [];
     layout = (fun a b -> ());
     display = (fun () -> ());
     mem = (fun s -> false);
     listener = (fun s -> false);}
  in
  dc.layout <- direct_layout dc;
  dc.mem <- in_rect dc;
  dc.display <- display_rect dc;
```



```
dc ;;
val create_component : int -> int -> component = <fun>
```

On définit alors le composant vide suivant :

```
# let empty_component = create_component 0 0 ;;
```

Celui-ci est utilisé comme valeur par défaut pour la construction de valeurs contenant au moins un composant comme, par exemple, une valeur de type *rich_status*.

Ajout de composants

La difficulté de l'ajout d'un composant à un conteneur est le positionnement du composant dans le conteneur. Le champ `layout` contient la fonction de placement. Elle prend un composant (un fils) et une liste d'options et calcule les nouvelles coordonnées du composant fils dans le conteneur. Différentes options peuvent être utilisées selon chaque fonction de placement. On détaille plusieurs fonctions de placement lors de la définition des conteneurs *panel* (voir *infra*, page ??). On décrit ici uniquement la mécanique de propagation de l'affichage, du changement de coordonnées et du traitement des événements. Cette propagation d'actions utilise intensivement la fonction `List.iter` qui applique une fonction à tous les éléments d'une liste.

La fonction `change_coord` applique un changement relatif de coordonnées à un composant et à tous ses fils.

```
# let rec change_coord c (dx,dy) =
  c.x <- c.x + dx; c.y <- c.y + dy;
  List.iter (fun s -> change_coord s (dx,dy) ) c.sons;;
val change_coord : component -> int * int -> unit = <fun>
```

La fonction `add_component` vérifie que les conditions d'ajout sont bien remplies, puis raccroche père (c) et fils (c1). La liste des options de placement est conservée dans le composant fils ce qui permet de la réutiliser en cas de modification de la règle de placement du père. La liste d'options passée à cette fonction

concerne les options de la fonction de placement. Il y a trois cas d'échec : le composant fils a déjà un père, le père n'est pas un conteneur et la taille du fils est trop grande.

```
# let add_component c cl lopt =  
  if cl.father <> [] then failwith "add_component : yet a  
  else  
    if not (is_container c ) then  
      failwith "add_component : not a container"  
    else  
      if (cl.x + cl.w > c.w) || (cl.y + cl.h > c.h)  
      then failwith "add_component : bad position"  
      else  
        c.layout cl lopt;  
        cl.layout_options <- lopt;  
        List.iter (fun s -> change_coord s (cl.x,cl.y)) cl.sons  
        c.sons <- cl::c.sons;  
        cl.father <- [c] ;;  
val add_component : component -> component -> lopt -> unit
```

Le retrait d'un composant à un niveau de l'arbre, réalisé par la fonction suivante, entraîne une modification des liens entre père et fils, mais aussi une modification des coordonnées du fils et de ses descendants :

```
# let remove_component c cl =  
  c.sons <- List.filter ((!=) cl) c.sons;  
  cl.father <- List.filter ((!=) c) cl.father;  
  List.iter (fun s -> change_coord s (- cl.x, - cl.y)) cl  
  cl.x <- 0; cl.y <- 0;;  
val remove_component : component -> component -> unit =
```

Le changement de la fonction de placement d'un conteneur dépend de l'existence de fils du conteneur. Dans le cas sans descendance, la modification est immédiate. Dans l'autre cas il faut d'abord enlever les fils du conteneur, modifier sa fonction de placement, puis rajouter les composants enlevés avec les mêmes options qu'au premier placement.

```
# let set_layout f c =  
  if c.sons = [] then c.layout <- f
```

```

else
let ls = c.sons in
List.iter (remove_component c) ls;
c.layout <- f;
List.iter (fun s -> add_component c s s.layout_options)
val set_layout : (component -> lopt -> unit) -> component

```

C'est pour cela que la liste d'options de placement d'un composant est conservée. Si la liste des options n'est pas reconnue par la nouvelle fonction de placement, celle-ci utilise les valeurs par défaut de ses options.

L'affichage d'un composant doit être propagé à ses fils pour les conteneurs. Un conteneur est affiché en arrière plan par rapport à ses fils. L'ordre d'affichage des fils importe peu car leur intersection est toujours vide.

```

# let rec display c =
  c.display ();
  List.iter (fun cx -> display cx ) c.sons;;
val display : component -> unit = <fun>

```

Cette fonction peut être directement appliquée à un composant pour le visualiser.

Gestion des événements

La gestion des événements physiques (clic souris, appui d'une touche, déplacement de la souris) utilise la fonction `Graphics.wait_next_event` (voir page ??) qui retourne un état physique (de type `Graphics.status`) suite à une action de l'utilisateur. Cet état physique est utilisé pour calculer un état enrichi (de type *rich_status*) contenant la nature de l'événement (type *rich_event*), l'état physique, les composants possédant le focus clavier et le focus général ainsi que le dernier composant ayant traité avec succès un tel état. Le focus général correspond à un composant prenant tous les événements.

On décrit les fonctions de manipulation des états enrichis, la propagation de ces états aux composants pour leur traitement, la création de ces états et la boucle principale de gestion des événements.

Fonctions sur les états

Les fonctions suivantes consultent les valeurs de positionnement souris et de focus. Les fonctions sur les focus nécessitent un paramètre supplémentaire : le composant gagnant ou perdant le focus.

```
# let get_event e = e.re;;
# let get_mouse_x e = e.stat.Graphics.mouse_x;;
# let get_mouse_y e = e.stat.Graphics.mouse_y;;
# let get_key e = e.stat.Graphics.key;;

# let has_key_focus e c = e.key_focus == c;;
# let take_key_focus e c = e.key_focus <- c;;
# let lose_key_focus e c = e.key_focus <- empty_componen
# let has_gen_focus e c = e.gen_focus == c;;
# let take_gen_focus e c = e.gen_focus <- c;;
# let lose_gen_focus e c = e.gen_focus <- empty_componen
```

Propagation des événements

Un état enrichi est envoyé à un composant pour son traitement. De la même manière que pour l'affichage, les composants fils ont priorité sur leur père pour le traitement d'un événement souris simple. Ils sont devant leur père et ils ne se recouvrent pas entre eux. Quand un composant reçoit l'état associé à un événement, il regarde s'il existe un descendant pour le traiter. Si c'est le cas, le descendant renvoie true et sinon, false. Dans ce dernier cas le composant ancêtre essaie de le traiter par la fonction de son champ listener.

Les états provenant d'une action clavier sont propagés différemment. Le composant ancêtre regarde s'il possède le focus clavier, si oui il traite l'événement et sinon. il le propage vers ses

fil.

Certains états sont produits par le traitement d'un événement premier (par exemple : la prise de focus par un composant entraîne sa perte par d'autres composants). De tels états sont traités immédiatement par le composant cible. Il en est de même pour les entrées et sorties de la souris de la zone d'un composant.

La fonction `send_event` prend une valeur de type *rich_status* et un composant, elle retourne un booléen indiquant si l'événement a été traité ou non.

```
# let rec send_event rs c =  
  match get_event rs with  
  | MouseDown | MouseUp | MouseDrag | MouseMove ->  
    if c.mem(get_mouse_x rs, get_mouse_y rs) then  
      if List.exists (fun sun -> send_event rs sun) c.sons then  
        else ( if c.listener rs then (rs.last <-c; true) else f  
      else false  
    | KeyPress | KeyRelease ->  
      if has_key_focus rs c then  
        ( if c.listener rs then (rs.last<-c; true)  
        else false )  
      else List.exists (fun sun -> send_event rs sun) c.sons  
    | _ -> c.listener rs;;  
val send_event : rich_status -> component -> bool = <fun
```

Il est à noter que la hiérarchie des composants est bien un arbre et non un graphe avec cycle. Ce qui garantit que la fonction récursive `send_event` ne bouclera pas.

Création d'événements

On différencie deux types d'événements : ceux produits par une action physique (comme un clic souris) et ceux qui découlent d'une action liée à l'historique (comme la sortie du curseur de la souris de la zone graphique d'un composant). On définira, en conséquence, deux fonctions de création d'états enrichis.

La fonction qui traite les premiers construit un état enrichi à partir

de deux états physiques :

```
# let compute_rich_event s0 s1 =
  if s0.Graphics.button <> s1.Graphics.button then (* boucle de
begin
  if s0.Graphics.button then MouseDown else MouseUp
end
  else if s1.Graphics.keypressed then KeyPress
  else if (s0.Graphics.mouse_x <> s1.Graphics.mouse_x )
    (s0.Graphics.mouse_y <> s1.Graphics.mouse_y ) then
begin
  if s1.Graphics.button then MouseDrag else MouseMove
end
  else raise Not_found;;
val compute_rich_event : Graphics.status -> Graphics.status
  <fun>
```

La fonction qui crée d'autres événements utilisent les deux derniers états enrichis :

```
# let send_new_events res0 res1 =
  if res0.key_focus <> res1.key_focus then
begin
  ignore(send_event {res1 with re = LostFocus} res0.key_focus)
  ignore(send_event {res1 with re = GotFocus} res1.key_focus)
end;
  if (res0.last <> res1.last) &&
    (( res1.re = MouseMove) || (res1.re = MouseDrag)) then
begin
  ignore(send_event {res1 with re = MouseExit} res0.last)
  ignore(send_event {res1 with re = MouseEnter} res1.last)
end;;
val send_new_events : rich_status -> rich_status -> unit
```

On définit une valeur initiale pour le type *rich_event*. Elle est employée pour initialiser l'historique de la boucle d'événements.

```
# let initial_re =
  { re = Exposure;
    stat = { Graphics.mouse_x=0; Graphics.mouse_y=0;
             Graphics.key = ' ';
```

```

Graphics.button = false;
Graphics.keypressed = false };
key_focus = empty_component;
gen_focus = empty_component;
last = empty_component } ;;

```

Boucle d'événements

La boucle d'événements gère l'ensemble des interactions sur un composant, en général le conteneur ancêtre de tous les composants de l'interface. Elle prend en entrée deux booléens, indiquant le réaffichage de l'interface après chaque traitement d'un événement physique (b_disp) et la gestion du déplacement de la souris (b_motion). Le dernier argument (OCnamec), est le composant principal, racine de l'arbre des composants.

```

# let loop b_disp b_motion c =
  let res0 = ref initial_re in
  try
    display c;
  while true do
    let lev = [Graphics.Button_down; Graphics.Button_up;
Graphics.Key_pressed] in
    let flev = if b_motion then (Graphics.Mouse_motion) ::
else lev in
    let s = Graphics.wait_next_event flev
    in
    let res1 = {!res0 with stat = s} in
    try
      let res2 = {res1 with
re = compute_rich_event !res0.stat res1.stat} in
      ignore(send_event res2 c);
      send_new_events !res0 res2;
      res0 := res2;
      if b_disp then display c
    with Not_found -> ()
    done
    with e -> raise e;;
val loop : bool -> bool -> component -> unit = <fun>

```

La seule manière de sortir de cette boucle est qu'une des actions de traitement d'un événement déclenche une exception.

Fonctions de test

On définit les deux fonctions suivantes, pour créer à la main des états correspondants aux événements souris et clavier.

```
# let make_click e x y =  
  {re = e;  
   stat = {Graphics.mouse_x=x; Graphics.mouse_y=y;  
   Graphics.key = ' '; Graphics.button = false;  
   Graphics.keypressed = false};  
   key_focus = empty_component;  
   gen_focus = empty_component;  
   last = empty_component}  
  
let make_key e ch c =  
  {re = e;  
   stat = {Graphics.mouse_x=0; Graphics.mouse_y=0;  
   Graphics.key = c; Graphics.button = false;  
   Graphics.keypressed = true};  
   key_focus = empty_component;  
   gen_focus = empty_component;  
   last = empty_component};;  
val make_click : rich_event -> int -> int -> rich_status  
val make_key : rich_event -> 'a -> char -> rich_status =
```

On pourra donc simuler l'envoi d'un événement souris pour les tests de l'interface en utilisant cette fonction.

Définition de composants

Les différents mécanismes d'affichage, de modification de coordonnées et de propagation d'événements sont en place. Il reste maintenant à définir des composants utiles et faciles à manipuler. On peut classer les composants en trois catégories :

- les composants simples ne traitant pas les événements (comme les textes à afficher);
- les composants simples traitant des événements, comme les champs de saisie;
- les conteneurs et les variantes de placement.

La communication de valeurs entre deux composants ou entre un composant et l'application s'effectue par modification physique d'une donnée partagée. Le partage est réalisé en créant des fermetures qui contiennent dans leur environnement les valeurs à modifier. De plus comme le comportement d'un composant peut varier à la suite du traitement d'un événement, ils conservent aussi dans leur fermeture de traitement un état interne. Par exemple la fonction de traitement d'une zone de saisie a accès au texte en cours d'écriture. Pour cela on écrit les composants de la manière suivante :

- définition d'un type pour l'état interne du composant;
- déclaration des fonctions de manipulation de cet état;
- écriture des fonctions d'affichage, d'appartenance d'un point au composant et de traitement des événements;
- réalisation de la fonction de création du composant qui associe ces fermetures aux champs du composant;
- test du composant en simulant le déclenchement d'événements.

Les fonctions de création prennent une liste d'options pour paramétrer le contexte graphique. Le calcul de la taille d'un composant à sa création a besoin d'utiliser le contexte graphique du composant sur la fenêtre graphique pour connaître la taille du texte à afficher.

On détaille l'implantation des composants suivants :

- texte simple (label)
- conteneur simple (panel)
- bouton simple (button)
- liste de choix (choice)
- champ de saisie (textfield)
- composant enrichi (border)

Composant label

Le composant le plus simple, que l'on appelle un *label*, affiche une chaîne de caractères à l'écran. Il ne traite pas d'événement. On décrit tout d'abord sa fonction d'affichage puis sa fonction de création.

L'affichage doit tenir compte des couleurs d'encre et de fond et des polices de caractères. C'est le rôle de la fonction `display_init` qui efface la zone graphique du composant et sélectionne la couleur de l'encre et la position du point courant. La fonction `display_label` affiche la chaîne passée en paramètre juste après l'appel à `display_init`.

```
# let display_init c =  
  Graphics.set_color (get_gc_bcol (get_gc c)); display_re  
  let gc= get_gc c in  
  use_gc gc;  
  let (a,b) = get_gc_cur gc in  
  Graphics.moveto (c.x+a) (c.y+b)  
  let display_label s c () =  
    display_init c; Graphics.draw_string s;;  
val display_init : component -> unit = <fun>  
val display_label : string -> component -> unit -> unit
```

Comme ce composant est fort simple, il n'est pas nécessaire de créer un état interne. Seule la fonction `display_label` connaît la chaîne à afficher qui est passée à la fonction de création.

```
# let create_label s lopt =  
  let gc = make_default_context () in set_gc gc lopt; use  
  let (w,h) = Graphics.text_size s in  
  let u = create_component w h in  
  u.mem <- (fun x -> false); u.display <- display_label  
  u.info <- "Label"; u.gc <- gc;  
  u;;  
val create_label : string -> (string * opt_val) list ->
```

Si l'on désire changer les couleurs de ce composant, il est nécessaire d'intervenir directement sur son contexte graphique.

L'affichage du *label*1 suivant est représenté à la figure 13.1.

```
# let courier_bold_24 = Sopt "*courier-bold-r-normal-*24
  and courier_bold_18 = Sopt "*courier-bold-r-normal-*18
# let l1 = create_label "Login : " ["Font", courier_bold_
  "Background", Copt gris1];;
```



Figure 13.1 : Affichage d'un *label*

Composant *panel*, conteneur et placement

On appelle *panel* un panneau graphique qui peut être un conteneur. La fonction de création par défaut est très simple, elle reprend principalement la fonction générale de création de composants en acceptant de plus un booléen indiquant s'il est un conteneur ou non. Les fonctions d'appartenance à la zone du *panel* et d'affichage sont celles par défaut de la fonction *create_component*.

```
# let create_panel b w h lopt =
  let u = create_component w h in
  u.container <- b;
  u.info <- if b then "Panel container" else "Panel";
  let gc = make_default_context () in set_gc gc lopt; u.g
  u;;
```

```
val create_panel :
  bool -> int -> int -> (string * opt_val) list -> compo
```

Le point délicat des conteneurs provient du placement des composants. On définit deux nouvelles fonctions de placement : *center_layout* et *grid_layout*. La première, *center_layout* place un composant au centre d'un conteneur :

```
# let center_layout c cl lopt =
```

```

  cl.x <- c.x + ((c.w -cl.w) /2); cl.y <- c.y + ((c.h -cl.h) /2);
  val center_layout : component -> component -> 'a -> unit

```

La seconde, `grid_layout` découpe un conteneur en grille où chaque case a la même taille. Les options de placement sont "Col" pour le numéro de la colonne et "Row" pour le numéro de la ligne.

```

# let grid_layout (a, b) c cl lopt =
  let px = theInt lopt "Col" 0
  and py = theInt lopt "Row" 0 in
  if (px >= 0) && (px < a) && (py >= 0) && (py < b) then
    let lw = c.w /a
    and lh = c.h /b in
    if (cl.w > lw) || (cl.h > lh) then
      failwith "grid_placement : too big component"
    else
      cl.x <- c.x + px * lw + (lw - cl.w)/2;
      cl.y <- c.y + py * lh + (lh - cl.h)/2;
    else failwith "grid_placement : bad position";;
val grid_layout :
  int * int -> component -> component -> (string * opt_val) -> unit
<fun>

```

Il est bien sûr possible d'en définir d'autres. On peut donc particulariser un conteneur en modifiant sa fonction de placement (`set_layout`). La figure 13.2 montre un *panel*, déclaré comme conteneur, dans lequel deux *labels* ont été ajoutés et correspond au programme suivant :

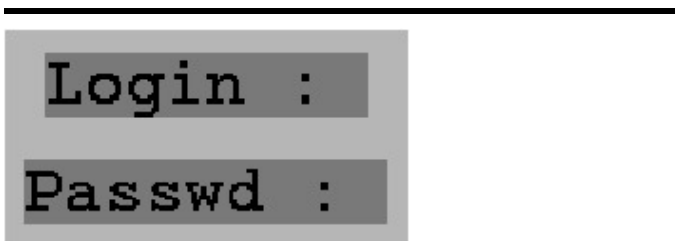


Figure 13.2 : Affichage d'un *panel*

```
# let l2 = create_label "Passwd : " ["Font", courier_bol
  "Background", Copt gris1] ;;
# let p1 = create_panel true 150 80 ["Background", Copt
# set_layout (grid_layout (1,2) p1) p1;;
# add_component p1 l1 ["Row", Iopt 1];;
# add_component p1 l2 ["Row", Iopt 0];;
```

Comme les composants ont besoin d'au moins un père pour s'intégrer dans l'interface, et que la bibliothèque Graphics ne possède qu'une seule fenêtre, on définit une fenêtre principale qui est un conteneur.

```
# let open_main_window w h =
  Graphics.close_graph();
  Graphics.open_graph (":0 "^(string_of_int w)^"x"^(string_of_int h));
  let u = create_component w h in
    u.container <- true;
    u.info <- "Fenêtre principale";
    u;;
val open_main_window : int -> int -> component = <fun>
```

Composant button

Un bouton simple ou *button* est un composant qui d'une part affiche un texte dans sa zone graphique et réagit au clic souris. Pour cela il conserve un état, valeur de type *button_state*, qui contient son texte et sa fonction de traitement du clic souris.

```
# type button_state =
  { txt : string; mutable action : button_state -> unit }
```

La création d'un état est effectuée par la fonction `create_bs`. La fonction `set_bs_action` modifie la fonction d'action et la fonction `get_bs_text` récupère le texte d'un bouton.

```
# let create_bs s = {txt = s; action = fun x -> ()}
let set_bs_action bs f = bs.action <- f
```

```

let get_bs_text bs = bs.txt;;
val create_bs : string -> button_state = <fun>
val set_bs_action : button_state -> (button_state -> unit) = <fun>
val get_bs_text : button_state -> string = <fun>

```

La fonction d'affichage est similaire à celle des *labels* à la différence près que le texte provient de l'état du bouton. La fonction d'écoute par défaut déclenche la fonction d'action par l'appui d'un bouton souris.

```

# let display_button c bs () =
  display_init c; Graphics.draw_string (get_bs_text bs)
  let listener_button c bs e = match get_event e with
    MouseDown -> bs.action bs; c.display (); true
  | _ -> false;;
val display_button : component -> button_state -> unit = <fun>
val listener_button : component -> button_state -> rich_text -> unit = <fun>

```

Tout est prêt pour définir la fonction de création de boutons simples :

```

# let create_button s lopt      =
  let bs = create_bs s in
  let gc = make_default_context () in
  set_gc gc lopt; use_gc gc;
  let w,h = Graphics.text_size (get_bs_text bs) in
  let u = create_component w h in
  u.display <- display_button u bs;
  u.listener <- listener_button u bs;
  u.info <- "Button";
  u.gc <- gc;
  u,bs;;
val create_button :
  string -> (string * opt_val) list -> component * button_state

```

Celle-ci retourne un couple dont le premier élément est le bouton créé et le deuxième l'état interne de ce bouton. Cela est particulièrement utile si l'on désire modifier la fonction d'action de celui-ci.

La figure 13.3 crée un *panel* auquel est ajouté un bouton. On associe une fonction d'action qui affiche la chaîne contenue dans le bouton sur la sortie standard.

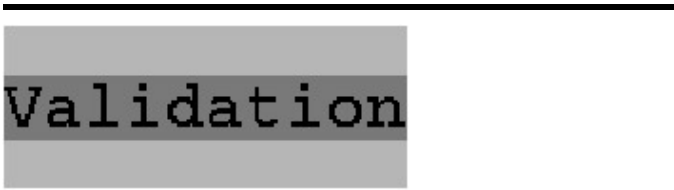


Figure 13.3 : Affichage d'un bouton et réaction au clic souris

```
# let b,bs = create_button "Validation" ["Font", courier
  "Background", Copt gris1];;
# let p2 = create_panel true 150 60 ["Background", Copt
# set_bs_action bs (fun bs -> print_string ( (get_bs_text
  print_newline()));;
# set_layout (center_layout p2) p2;;
# add_component p2 b [];;
```

À la différence des *labels*, un composant bouton sait réagir à un événement souris. Pour le tester on envoie l'événement << clic souris >> en position centrale du *panel* *p2*, qui est répercuté au bouton, ce qui déclenche l'action associée à ce dernier :

```
# send_event (make_click MouseDown 75 30) p2;;
Validation...
- : bool = true
```

et retourne la valeur *true* indiquant que l'événement a bien été traité.

Composant *choice*

Les listes de choix (ou *choice*) permet de sélectionner un des choix proposés par un clic souris. Il existe toujours un choix courant. Un clic souris sur un autre choix fait changer le choix courant et déclenche une action. On utilise la même technique que pour les boutons simples. On commence par définir l'état d'une liste de choix :

```
# type choice_state =  
  { mutable ind : int; values : string array; mutable sep  
    mutable height : int; mutable action : choice_state ->
```

L'indice ind indique la chaîne à marquer rangée dans values. Les champs sep et height exprimés en pixels correspondent respectivement à la séparation entre deux choix et à la hauteur de chaque choix. La fonction d'action prend en entrée un tel état et peut s'aiguiller selon l'indice.

On définit maintenant la fonction de création d'un état et la fonction de modification du traitement.

```
# let create_cs sa = {ind = 0; values = sa; sep = 2;  
  height = 1; action = fun x -> ()}  
  let set_cs_action cs f = cs.action <- f  
  let get_cs_text cs = cs.values.(cs.ind);;  
val create_cs : string array -> choice_state = <fun>  
val set_cs_action : choice_state -> (choice_state -> uni  
val get_cs_text : choice_state -> string = <fun>
```

L'affichage donne la liste de tous les choix et souligne le choix courant en inverse vidéo. La fonction de traitement des événements déclenche la fonction de traitement de l'état au relâchement du bouton de souris.

```
# let display_choice c cs () =  
  display_init c;  
  let (x,y) = Graphics.current_point()  
  and nb = Array.length cs.values in  
  for i = 0 to nb-1 do  
    Graphics.moveto x (y + i*(cs.height+ cs.sep));  
    Graphics.draw_string cs.values.(i)
```



```

done;
Graphics.set_color (get_gc_fcol (get_gc c));
Graphics.fill_rect x (y+ cs.ind*(cs.height+ cs.sep)) c.
Graphics.set_color (get_gc_bcol (get_gc c));
Graphics.moveto x (y + cs.ind*(cs.height + cs.sep));
Graphics.draw_string cs.values.(cs.ind) ;;
val display_choice : component -> choice_state -> unit -

# let listener_choice c cs e = match e.re with
  MouseUp ->
    let x = e.stat.Graphics.mouse_x
    and y = e.stat.Graphics.mouse_y in
    let cy = c.y in
    let i = (y - cy) / ( cs.height + cs.sep) in
    cs.ind <- i; c.display ();
    cs.action cs; true
  | _ -> false ;;
val listener_choice : component -> choice_state -> rich_
  <fun>

```

La création d'une liste de choix prend une liste de chaînes de caractères et une liste d'options pour retourner le composant ainsi que son état.

```

# let create_choice lc lopt =
  let sa = (Array.of_list (List.rev lc)) in
  let cs = create_cs sa in
  let gc = make_default_context () in
  set_gc gc lopt; use_gc gc;
  let awh = Array.map (Graphics.text_size) cs.values in
  let w = Array.fold_right (fun (x,y) -> max x) awh 0
  and h = Array.fold_right (fun (x,y) -> max y) awh 0 in
  let h1 = (h+cs.sep) * (Array.length sa) + cs.sep in
  cs.height <- h;
  let u = create_component w h1 in
  u.display <- display_choice u cs;
  u.listener <- listener_choice u cs ;
  u.info <- "Choice " ^ (string_of_int (Array.length cs.values))
  u.gc <- gc;
  u,cs;;

```

```
val create_choice :
  string list -> (string * opt_val) list -> component * c
```

Les trois images de la figure 13.4 montre un *panel* auquel a été ajouté une liste de choix. On associe une fonction d'action qui affiche sur la sortie standard la chaîne correspondant au choix indiqué dans le button. Chaque image fait suite à un clic souris simulé par le programme suivant.

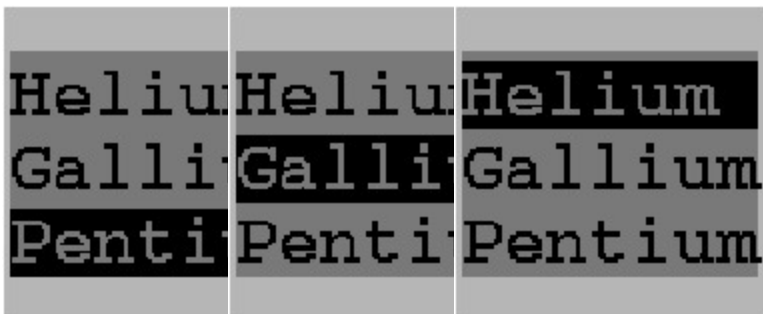


Figure 13.4 : Affichage d'une liste de choix et sélection

```
# let c,cs = create_choice ["Helium"; "Gallium"; "Pentium"]
  ["Font", courier_bold_24;
   "Background", Copt gris1];;
# let p3 = create_panel true 110 110 ["Background", Copt gris1];;
# set_cs_action cs (fun cs -> print_string ( (get_cs_text cs) ^ "\n");;
# set_layout (center_layout p3) p3;;
# add_component p3 c [];
```

Là aussi nous pouvons le tester directement en envoyant plusieurs événements. Cet envoi modifie la sélection, comme dans l'image du centre de la figure 13.4.

```
# send_event (make_click MouseUp 60 55 ) p3;;
```

```
Gallium...  
- : bool = true
```

De même l'événement suivant envoyé sélectionne le premier élément de la liste de choix

```
# send_event (make_click MouseUp 60 90 ) p3;;  
Helium...  
- : bool = true
```

Composant *textfield*

Un champ de saisie, ou *textfield*, est une zone permettant d'écrire du texte sur une ligne. Le texte peut s'afficher de gauche à droite ou de droite à gauche comme pour une calculatrice. De plus un curseur indique la position du prochain caractère à entrer. On définit un état un peu plus complexe. Il comprend le texte en cours de saisie, la direction de la saisie, l'information sur le curseur, l'information sur l'affichage des caractères, et la fonction d'action.

```
# type textfield_state =  
  { txt : string;  
    dir : bool; mutable ind1 : int; mutable ind2 : int; len : int;  
    mutable visible_cursor : bool; mutable cursor : char;  
    mutable visible_echo : bool; mutable echo : char;  
    mutable action : textfield_state -> unit } ;;
```

La création d'un tel état demande le texte initial, le nombre de caractères de la zone de saisie et la direction de saisie.

```
# let create_tfs txt size dir =  
  let l = String.length txt in  
  (if size < l then failwith "create_tfs");  
  let ind1 = if dir then 0 else size-1-1  
  and ind2 = if dir then l else size-1 in  
  let n_txt = (if dir then (txt^(String.make (size-1) ' '))  
    else ((String.make (size-1) ' ')^txt)) in  
  {txt = n_txt; dir=dir; ind1 = ind1; ind2 = ind2; len=size;  
    visible_cursor = false; cursor = ' '; visible_echo =  
    action= fun x -> ()};;
```

```
val create_tfs : string -> int -> bool -> textfield_state
```

Les fonctions suivantes permettent de modifier certains champs et de retourner le texte affiché.

```
# let set_tfs_action tfs f = tfs.action <- f
let set_tfs_cursor b c tfs = tfs.visible_cursor <- b;
let set_tfs_echo b c tfs = tfs.visible_echo <- b; tfs.
let get_tfs_text tfs =
if tfs.dir then String.sub tfs.txt tfs.ind1 (tfs.ind2 -
else String.sub tfs.txt (tfs.ind1+1) (tfs.ind2 - tfs.in
```

La fonction `set_tfs_text` modifie le text de l'état `tfs` du composant `tf` par la chaîne `txt`.

```
# let set_tfs_text tf tfs txt =
let l = String.length txt in
if l > tfs.len then failwith "set_tfs_text";
String.blit (String.make tfs.len ' ') 0 tfs.txt 0 tfs.
if tfs.dir then (String.blit txt 0 tfs.txt 0 l;
tfs.ind2 <- l )
else ( String.blit txt 0 tfs.txt (tfs.len -l) l;
tfs.ind1 <- tfs.len-l-1 );
tf.display ();;
val set_tfs_text : component -> textfield_state -> string
```

L'affichage tient compte des informations sur l'écho des caractères et sur la visibilité du curseur. La fonction `display_textfield` appelle la fonction `display_cursor` qui souligne la position du curseur.

```
# let display_cursor c tfs =
if tfs.visible_cursor then
( use_gc (get_gc c);
let (x,y) = Graphics.current_point() in
let (a,b) = Graphics.text_size " " in
let shift = a * (if tfs.dir then max (min (tfs.len-1)
else tfs.ind2) in
Graphics.moveto (c.x+x + shift) (c.y+y);
Graphics.draw_char tfs.cursor);;
val display_cursor : component -> textfield_state -> unit
# let display_textfield c tfs () =
```

```

display_init c;
let s = String.make tfs.len ' '
and txt = get_tfs_text tfs in
let nl = String.length txt in
if (tfs.indl >= 0) && (not tfs.dir) then <

```

Recherche de chemins de moindre coût

La recherche de chemin de moindre coût dans un graphe orienté est utilisée dans de nombreuses applications. On s'intéresse ici à la détermination d'un itinéraire de voyage. Pour cela on représente les différentes étapes et routes par un graphe orienté, valué par le coût entre deux étapes. On utilise l'algorithme de Dijkstra pour la détermination du chemin de moindre coût entre deux étapes. Cette application va être l'occasion d'utiliser les bibliothèques présentées précédemment. On peut citer dans l'ordre d'apparition, les modules Genlex et Printf pour les entrées-sorties, le module Weak pour implanter un cache, le module Sys pour compter le temps gagné par le cache et la bibliothèque Upi pour construire l'interface graphique. On réutilise le module Sys à la construction d'une application autonome en récupérant le nom du fichier de description du graphe comme argument de la ligne de commande.

Représentation des graphes

Un graphe orienté valué est défini par son ensemble de sommets, son ensemble d'arcs et l'ensemble des valeurs associées à chaque arc. Il existe de nombreuses implantations du type de données *graphe orienté valué*.

- les matrices d'adjacence :
où chaque case de la matrice ($m(i,j)$) représente un arc entre les sommets i et j , son contenu est la valeur associée à l'arc ;
- les listes d'adjacence :
où chaque sommet i est lié à une liste [$j_1; \dots; j_n$] de sommets et chaque couple (i, j_k) est un arc du graphe ;
- un triplet :
liste de sommets, liste des d'arcs et fonction de calcul des valeurs des arcs.

La pertinence de ces représentations varie selon la taille du graphe et le nombre d'arcs. Comme le propos de cette application est en partie de montrer comment mémoriser certains calculs déjà effectués sans remplir la mémoire, on utilise une matrice d'adjacence pour les graphes orientés valués. Ainsi l'occupation mémoire ne sera pas perturbée par la manipulation de listes.

```
# type cout    = Nan | Cout of float;;  
# type mat_adj = cout array array;;  
# type 'a graphe = { mutable ind : int;  
  taille : int;  
  sommets : 'a array;  
  m : mat_adj};;
```

Le champ `taille` contient le nombre maximal de sommets et le champ `ind`, le nombre courant de sommets.

On définit ensuite quelques fonctions permettant de créer pas à pas un arc.

La fonction de création d'un graphe prend en arguments un sommet et le nombre maximal de sommets.

```
# let cree_graphe s t =  
  { ind = 0; taille = t; sommets = Array.create t s;  
    m = Array.create_matrix t t Nan } ;;  
val cree_graphe : 'a -> int -> 'a graphe = <fun>
```

La fonction `appartient` vérifie si un sommet `n` est dans le graphe `g`.

```
# let appartient s g =  
  let rec aux i =  
    (i < g.taille) & ((g.sommets.(i) = s) or (aux (i+1)))  
  in aux 0;;  
val appartient : 'a -> 'a graphe -> bool = <fun>
```

La fonction `index` retourne l'indice d'un sommet `s` dans un graphe `g`. Si ce sommet n'existe pas alors l'exception `Not_found` est déclenchée.

```
# let index s g =
  let rec aux i =
    if i >= g.taille then raise Not_found
    else if g.sommets.(i) = s then i
    else aux (i+1)
  in aux 0 ;;
val index : 'a -> 'a graphe -> int = <fun>
```

Les deux fonctions suivantes ajoutent respectivement un sommet et un arc entre deux sommets de coût c à un graphe.

```
# let ajoute_sommet s g =
  if g.ind = g.taille then failwith "le graphe est plein"
  else if appartient s g then failwith "le sommet existe"
  else (g.sommets.(g.ind) <- s; g.ind <- g.ind + 1) ;;
val ajoute_sommet : 'a -> 'a graphe -> unit = <fun>
# let ajoute_arc s1 s2 c g =
  try
    let x = index s1 g and y = index s2 g in
    g.m.(x).(y) <- Cout c
  with Not_found -> failwith "sommet inexistant" ;;
val ajoute_arc : 'a -> 'a -> float -> 'a graphe -> unit
```

On peut alors facilement créer un graphe orienté valué complet à partir d'une liste de sommets et d'arcs. La fonction `test_aho` construit le graphe de la figure [13.8](#) :

```
# let test_aho () =
  let g = cree_graphe "rien" 5 in
  List.iter (fun x -> ajoute_sommet x g) ["A"; "B"; "C";
  List.iter (fun (a,b,c) -> ajoute_arc a b c g)
    ["A","B",10.;
    "A","D",30.;
    "A","E",100.0;
    "B","C",50.;
    "C","E",10.;
    "D","C",20.;
    "D","E",60.];
  for i=0 to g.ind -1 do g.m.(i).(i) <- Cout 0.0 done;
  g;;
val test_aho : unit -> string graphe = <fun>
```

```
# let a = test_aho();;
val a : string graphe =
  {ind=5; taille=5; sommets=[|"A"; "B"; "C"; "D"; "E"|];
    m=[| | | Cout 0; Cout 10; Nan; Cout 30; Cout ...|]; ...|]}
```

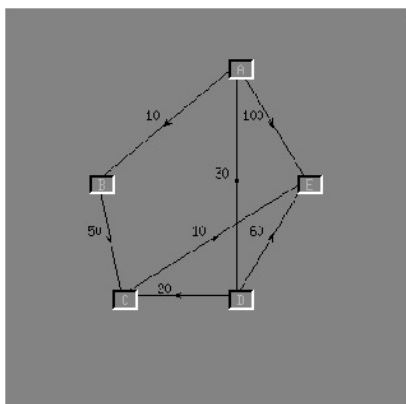


Figure 13.8 : Graphe de test

Construction de graphes

Il peut être assez fastidieux de devoir construire les graphes directement dans un programme. Pour éviter cela, nous définissons un format pour représenter les graphes sous une forme textuelle. On peut alors lire, à partir d'un fichier texte, les informations d'un graphe et le construire.

Un fichier de description d'un graphe contient une suite de lignes. Chaque ligne suit un des formats suivants :

- le nombre de noeuds : TAILLE nombre
- le nom d'un sommet : SOMMET nom
- le coût d'un arc : ARC nom1 nom2 cout
- des commentaires : # phrase

Par exemple, le fichier aho.dat suivant décrit le graphe de la figure 13.8 :

```
TAILLE 5
SOMMET A
SOMMET B
SOMMET C
SOMMET D
SOMMET E
ARC A B 10.0
ARC A D 30.0
ARC A E 100.0
ARC B C 50.
ARC C E 10.
ARC D C 20.
ARC D E 60.
```

Pour analyser de tels fichiers, on utilise le module Genlex d'analyse lexicale. L'analyseur lexicale est construit à partir de la liste de mots clés keywords.

La fonction `parser_line` effectue les actions associées aux mots clés en modifiant une référence sur un graphe.

```
# let keywords = [ "TAILLE"; "SOMMET"; "ARC"; "#" ];;
val keywords : string list = ["TAILLE"; "SOMMET"; "ARC";
# let lexer_line l = Genlex.make_lexer keywords (Stream.
val lexer_line : string -> Genlex.token Stream.t = <fun>
# let parser_line g s = match s with parser
  [< ' (Genlex.Kwd  "TAILLE"); ' (Genlex.Int n) >] ->
    g := cree_graphe "" n
  | [< ' (Genlex.Kwd  "SOMMET"); ' (Genlex.Ident nom) >] ->
    ajoute_sommet nom !g
  | [< ' (Genlex.Kwd  "ARC"); ' (Genlex.Ident n1);
    ' (Genlex.Ident  n2); ' (Genlex.Float c) >] ->
    ajoute_arc n1 n2 c !g
  | [< ' (Genlex.Kwd  "#") >] -> ()
  | [<>] -> () ;;
val parser_line : string graphe ref -> Genlex.token Stre
```

On utilise cet analyseur pour définir la fonction de création d'un

graphe utilise à partir des données d'un fichier :

```
# let create_graphe name =  
  let g = ref {ind=0; taille=0; sommets=[|]|}; m = [|]|}  
  let ic = open_in name in  
  try  
    print_string ("Loading "^name^": ");  
    while true do  
      print_string ".";  
      let l = input_line ic in parser_line g (lexer_line l)  
    done;  
    !g  
  with End_of_file -> print_newline(); close_in ic; !g ;;  
val create_graphe : string -> string graphe = <fun>
```

L'appel suivant construit un graphe à partir du fichier aho.dat.

```
# let b = create_graphe "PROGRAMMES/aho.dat" ;;  
Loading PROGRAMMES/aho.dat: .....  
val b : string graphe =  
  {ind=5; taille=5; sommets=[|"A"; "B"; "C"; "D"; "E"|];  
  m=[|]|Nan; Cout 10; Nan; Cout 30; Cout 100|]; ...|}]
```

Algorithme de Dijkstra

L'algorithme de Dijkstra permet de trouver le chemin de coût minimal entre deux sommets. Le coût d'un chemin entre un sommet s_1 et un sommet s_2 est la somme des coûts des arcs de ce chemin. Un coût est toujours positif, c'est-à-dire que l'on ne gagne pas de repasser par un sommet. C'est une condition de correction de l'algorithme.

L'algorithme de Dijkstra calcule en fait tous les chemins de moindre coût d'une source S_1 vers l'ensemble sommets atteignables depuis cette source. L'idée est de supposer connus les chemins de moindre coût de S_1 vers un ensemble de sommets et d'étendre cet ensemble en considérant les sommets accessibles par un arc à partir de l'un

des sommets déjà connu en retenant ceux qui augmentent le moins le coût des chemins.

Pour conserver l'état de la recherche, on définit le type *etat_recherche* puis une fonction de création de l'état initial :

```
# type etat_recherche = { chemins : int array;  
  deja_traites : bool array;  
  distances : cout array;  
  source :int;  
  nb : int};;  
# let creer_etat () = { chemins = [| |]; deja_traites =  
  nb = 0; source = 0};;
```

Le champ *source* contient le sommet de départ ; le champ *deja_traites*, les sommets dont on connaît déjà le chemin optimal à partir de la source et le champ *nb*, le nombre de sommets déjà traités. Le vecteur *distances* tient à jour les distances minimales entre la source et les autres sommets. Le vecteur *chemin* contient, pour chaque sommet, le sommet le précédent dans le chemin de moindre coût. On peut ainsi reconstituer un chemin de la source vers n'importe quel autre sommet.

Fonctions sur les coûts

On définit tout d'abord quatre fonctions utilitaires sur les coûts : *a_cout* pour tester de l'existence d'un arc, *float_of_cout* pour la conversion en flottant, *add_cout* pour l'addition de deux coûts et *inf_cout* vérifiant si un coût est inférieur à un autre.

```
# let a_cout c = match c with Nan -> false | _-> true;;  
val a_cout : cout -> bool = <fun>  
# let float_of_cout c = match c with  
  Nan -> failwith "float_of_cout"  
  | Cout x -> x;;  
val float_of_cout : cout -> float = <fun>  
# let add_cout c1 c2 = match (c1,c2) with  
  Cout x, Cout y -> Cout (x+y)  
  | Nan, Cout y -> c2
```

```

| Cout x, Nan -> c1
| Nan, Nan -> c1;;
val add_cout : cout -> cout -> cout = <fun>
# let inf_cout c1 c2 = match (c1,c2) with
  Cout x, Cout y -> x < y
  | Cout x, Nan -> true
  | _, _ -> false;;
val inf_cout : cout -> cout -> bool = <fun>

```

La valeur Nan joue un rôle particulier dans les calculs et la comparaison. Nous y reviendrons lorsque nous aurons donné la fonction principale (voir page ??).

Implantation de l'algorithme

La recherche d'un sommet servant à étendre les chemins déjà calculés est divisée en deux fonctions : `premier_non_traite` qui sélectionne un premier sommet non encore visité ; celui-ci sert de valeur initiale pour la fonction `pp_non_traite` qui retourne le sommet non encore traité dont l'ajout à l'ensemble des chemin construit est minimal.

```

# exception Trouve of int;;
exception Trouve of int
# let premier_non_traite er =
  try
    for i=0 to er.nb-1 do
      if not er.deja_traites.(i) then raise (Trouve i)
    done;
    raise Not_found;
  0
  with Trouve i -> i ;;
val premier_non_traite : etat_recherche -> int = <fun>
# let pp_non_traite p er =
  let si = ref p
  and sd = ref er.distances.(p) in
  for i=p+1 to er.nb-1 do
    if not er.deja_traites.(i) then

```

```

if inf_cout er.distances.(i)  !sd then
  ( sd := er.distances.(i);
  si := i )
done;
!si,!sd;;
val pp_non_traite : int -> etat_recherche -> int * cout

```

La fonction une_etape sélectionne un nouveau sommet, l'ajoute à l'ensemble des sommets traités et met à jour les chemins et distances des autres sommets non traités s'il y a lieu.

```

# exception No_way;;
exception No_way
# let une_etape er  g =
  let p = premier_non_traite er in
  let np,nc = pp_non_traite p er in
  if not(a_cout nc  ) then raise No_way
  else
    begin
      er.deja_traites.(np) <- true;
      for i = 0 to er.nb -1 do
        if not er.deja_traites.(i) then
          if a_cout g.m.(np).(i) then
            let ic = add_cout er.distances.(np)  g.m.(np).(i) in
            if inf_cout ic er.distances.(i) then (
              er.chemins.(i) <- np;
              er.distances.(i) <- ic
            )
          done;
        er
      end;;
val une_etape : etat_recherche -> 'a graphe -> etat_rech

```

Il ne reste plus qu'à itérer la fonction précédente pour implanter l'algorithme de Dijkstra. La fonction dij prend un sommet et un graphe puis retourne une valeur de type etat_recherche contenant les informations pour déduire le chemin de moindre coût de cette source vers n'importe quel sommet atteignable à partir de celle-ci.

```

# let dij s g =
  if appartient s g then

```

```

begin
let i = index s g in
let er = { chemins = Array.create g.ind (-1) ;
deja_traites = Array.create g.ind false;
distances = Array.create g.ind Nan;
nb = g.ind;
source = i} in
er.deja_traites.(i) <- true;
for j=0 to g.ind-1 do
let c = g.m.(i).(j) in
er.distances.(j) <- c;
if a_cout c then er.chemins.(j) <- i
done;
try
for k = 0 to er.nb-2 do
ignore(une_etape er g)
done;
er
with No_way -> er
end
else failwith "dij: sommet inconnu";;
val dij : 'a -> 'a graphe -> etat_recherche = <fun>

```

La valeur Nan est donnée comme valeur initiale des distances. Elle représente une distance infinie, ce qui est cohérent avec la fonction de comparaison `inf_cout`. En revanche. Pour l'addition des coûts (fonction `add_cout`, cette valeur est considérée comme nulle. Ce qui permet une mise à jour simple du tableau des distances.

On peut donc maintenant tester la recherche selon l'algorithme de Dijkstra.

```

# let a = test_aho ();;
# let r = dij "A" a;;

```

Les résultats bruts sont les suivants :

```

# r.chemins;;
- : int array = [|0; 0; 3; 0; 2|]
# r.distances;;
- : cout array = [|Cout 0; Cout 10; Cout 50; Cout 30; Cout 0|]

```

Pour rendre ces résultats plus lisibles, nous définissons ci-dessous une fonction d'affichage.

Affichage de la solution

En utilisant le tableau chemins de la valeur calculée par la fonction dij, on n'a que le dernier arc des chemins calculés. Il faut donc, pour afficher la totalité de ces chemins, remonter récursivement jusqu'à la source.

```
# let aff_etat f (g,et) dest =  
  if appartient dest g then  
    let d = index dest g in  
    let rec aux is =  
      if is = et.source then Printf.printf "%a" f g.sommets.  
    else (  
      let old = et.chemins.(is) in  
      aux old;  
      Printf.printf " -> (%4.1f) %a" (float_of_cout g.m.(old)  
      f g.sommets.(is)  
    )  
  in  
    if not(a_cout et.distances.(d)) then Printf.printf "no  
  else (  
    aux d;  
    Printf.printf " = %4.1f\n" (float_of_cout et.distances.  
val aff_etat :  
  (out_channel -> 'a -> unit) -> 'a graphe * etat_recherch  
<fun>
```

Cette fonction récursive utilise la pile d'appel pour afficher dans le bon ordre les sommets. On note l'utilisation d'un format "a" pour conserver le polymorphisme des graphes y compris pour l'affichage, d'où le paramètre fonctionnel f.

Le chemin optimal entre le sommet "A" (indice 0) et "E" (indice 4) est affiché ainsi :

```
# aff_etat (fun x y -> Printf.printf "%s!" y) (a,r) "E";
```

```
A! -> (30.0) D! -> (20.0) C! -> (10.0) E! = 60.0  
- : unit = ()
```

On indique les différents sommets du chemin ainsi que les coûts de chaque étape.

Utilisation d'un cache

L'algorithme de Dijkstra calcule tous les chemins de moindre coût issus d'une source. On aimerait donc conserver ces valeurs en vue d'une prochaine recherche à partir de la même source. Cette mémorisation de résultats risque néanmoins d'occuper une part importante de la mémoire. D'où l'idée d'utiliser les << pointeurs faibles >>. Si on conserve les résultats des recherches à partir d'une source dans un tableau de pointeurs faibles, il devient ensuite possible, pour une nouvelle recherche, de vérifier si le calcul a déjà été effectué. Comme ce sont des pointeurs faibles, l'espace mémoire de ces états peut être libéré en cas de besoin par le GC. Cela permet de ne pas perturber le reste du programme par une occupation mémoire trop importante. Au pire des cas le calcul peut être refait s'il y a une demande.

Implantation du cache

On définit un nouveau type *'a recherche_graphe* :

```
# type 'a recherche_graphe =  
  { g : 'a graphe; w : etat_recherche Weak.t } ;;
```

Les champs *g* et *w* correspondent au graphe et au tableau de pointeurs faibles sur les états de recherche pour chaque source possible.

On construit de telles valeurs par la fonction *create_recherche*.

```
# let create_rech_graphe g =  
  { g = g;  
    w = Weak.create g.ind } ;;
```



```
val create_rech_graphe : 'a graphe -> 'a recherche_graphe
```

La fonction `dij_rapide` vérifie si la recherche a déjà été calculée, si oui, elle retourne le résultat mémorisé, sinon, elle effectue le calcul et l'enregistre dans le tableau de pointeurs faibles.

```
# let dij_rapide s rg =  
  let i = index s rg.g in  
  match Weak.get rg.w i with  
  None -> let er = dij s rg.g in  
  Weak.set rg.w i (Some er);  
  er  
  | Some er -> er;;  
val dij_rapide : 'a -> 'a recherche_graphe -> etat_reche
```

La fonction d'affichage reste utilisable :

```
# let rg_a = create_rech_graphe a in  
  let r = dij_rapide "A" rg_a in  
  aff_etat (fun x y -> Printf.printf "%s!" y) (a,r) "E" ;  
A! -> (30.0) D! -> (20.0) C! -> (10.0) E! = 60.0  
- : unit = ()
```

Évaluation des performances

Nous allons tester les performances des fonctions `dij` et `dij_rapide` en itérant chacune d'elle sur une liste de source tirée au hasard. On simule ainsi une application dans laquelle il faudrait calculer souvent un chemin entre deux points d'un graphe (un itinéraire ferroviaire, par exemple).

Pour obtenir les temps de calcul, nous définissons la fonction suivante :

```
# let exe_time f g ss =  
  let t0 = Sys.time() in  
  Printf.printf "Début (%5.2f)\n" t0;  
  List.iter (fun s -> ignore(f s g)) ss;  
  let t1 = Sys.time() in  
  Printf.printf "Fin (%5.2f)\n" t1;
```

```
Printf.printf "Durée = (%5.2f)\n" (t1 -. t0) ;;
val exe_time : ('a -> 'b -> 'c) -> 'b -> 'a list -> unit
```

On tire donc au hasard une liste de 20000 sommets et on mesure les performances contenues sur le graphe a :

```
# let ss =
  let ss0 = ref [] in
  let i0 = int_of_char 'A' in
  let new_s i = Char.escaped (char_of_int (i0+i)) in
  for i=0 to 20000 do ss0 := (new_s (Random.int a.taille))
  !ss0 ;;
val ss : string list =
  ["A"; "B"; "D"; "A"; "E"; "C"; "B"; "B"; "D"; "E"; "B";
  "D"; "D"; "A"; "E"; ...]
# Printf.printf"Fonction dij :\n";
  exe_time dij a ss ;;
Fonction dij :
Début ( 7.08)
Fin ( 8.02)
Durée = ( 0.94)
- : unit = ()
# Printf.printf"Fonction dij_rapide :\n";
  exe_time dij_rapide (create_rech_graphe a) ss ;;
Fonction dij_rapide :
Début ( 8.02)
Fin ( 8.13)
Durée = ( 0.11)
- : unit = ()
```

Les résultats obtenus sont cohérents. Il est clair que l'accès direct au résultat contenu dans le cache est bien plus rapide que la reconstruction du résultat.

Interface graphique

On se propose de construire l'interface graphique de visualisation des graphes en utilisant la bibliothèque Upi. Cette interface permet de sélectionner les sommets origine et destination du chemin à rechercher. Une fois le chemin de moindre coût trouvé, celui-ci

s'affiche graphiquement. On définit le type 'a gg qui regroupe les champs de description de graphe et de la recherche, ainsi que les champs de l'interface graphique.

```
# #load "PROGRAMMES/upi.cmo";;

# type 'a gg = { mutable src : 'a * Upi.component;
  mutable dest : 'a * Upi.component;
  pos : (int * int) array;
  rg : 'a recherche_graphe;
  mutable etat : etat_recherche;
  mutable main : Upi.component;
  to_string : 'a -> string;
  from_string : string -> 'a } ;;
```

Les champs src et dest sont des couples (sommet,composant) liant un sommet à un composant. Le champ pos contient les positions de chaque composant. Le champ main est le conteneur principal de l'ensemble des composants. Les deux fonctions to_string et from_string sont les fonctions de conversion du type 'a avec les chaînes. Les éléments nécessaires pour la construction de telles valeurs sont les informations du graphe, le tableau de positions et les fonctions de conversion.

```
# let cree_gg rg vpos ts fs =
  {src = rg.g.sommets.(0), Upi.empty_component;
  dest = rg.g.sommets.(0), Upi.empty_component;
  pos = vpos;
  rg = rg;
  etat = creer_etat () ;
  main = Upi.empty_component;
  to_string = ts;
  from_string = fs};;
val cree_gg :
  'a recherche_graphe ->
  (int * int) array -> ('a -> string) -> (string -> 'a) -
```

Visualisation

Le dessin du graphe nécessite de dessiner les sommets et de tracer les arcs. Les sommets sont représentés par des composants boutons de la bibliothèque Upi. Par contre les arcs sont directement tracés dans la fenêtre principale. La fonction `display_arc` affiche les différents arcs. La fonction `display_shortest_path` affiche dans une autre couleur le chemin trouvé.

Tracé des arcs

Un arc relie deux sommets et possède une valeur associée. Le lien entre deux sommets peut être représenté par un tracé de ligne. La difficulté principale provient de l'affichage du sens de cette ligne. On choisit de le représenter par une flèche. Pour être bien orientée, celle-ci doit subir une rotation de l'angle que fait la ligne avec l'axe des abscisses. Enfin les coûts doivent être affichés à côté de l'arc.

Pour le tracé d'une flèche d'un arc, on définit : les fonctions `rotate` et `translate` qui effectuent respectivement une rotation et une translation ; la fonction `display_arrow` dessine la flèche dont l'extrémité est en fait un triangle.

```
# let rotate l a =
  let ca = cos a and sa = sin a in
  List.map (function (x,y) -> ( x*.ca +. -.y*.sa, x*.sa +. y*.ca )) l
val rotate : (float * float) list -> float -> (float * float) list
# let translate l (tx,ty) =
  List.map (function (x,y) -> (x +. tx, y +. ty)) l;;
val translate : (float * float) list -> float * float -> (float * float) list
# let display_arrow (mx,my) a =
  let triangle = [(5.,0.); (-3.,3.); (1.,0.); (-3.,-3.); (5.,0.)] in
  let tr = rotate triangle a in
  let ttr = translate tr (mx,my) in
  let tt = List.map (function (x,y) -> (int_of_float x, int_of_float y)) ttr in
  Graphics.fill_poly (Array.of_list tt);;
val display_arrow : float * float -> float -> unit = <fu
```

L'affichage du coût d'un arc positionne le point où démarre le texte

en fonction de l'angle de l'arc.

```
# let display_label (mx,my) a lab =  
  let (sx,sy) = Graphics.text_size lab in  
  let pos = [ float(-sx/2),float(-sy) ] in  
  let pr = rotate pos a in  
  let pt = translate pr (mx,my) in  
  let px,py = List.hd pt in  
  let ox,oy = Graphics.current_point () in  
  Graphics.moveto ((int_of_float mx)-sx-6)  
    ((int_of_float my) );  
  Graphics.draw_string lab;  
  Graphics.moveto ox oy;;  
val display_label : float * float -> float -> string ->
```

L'affichage d'un arc reprend les fonctions précédentes. Ses paramètres sont le graphe interfacé gg, les sommets i et j, et la couleur du tracé (col).

```
# let display_arc gg col i j =  
  let g = gg.rg.g in  
  let x,y = gg.main.Upi.x,gg.main.Upi.y in  
  if a_cout g.m.(i).(j) then (  
    let (a1,b1) = gg.pos.(i)  
    and (a2,b2) = gg.pos.(j) in  
    let x0,y0 = x+a1,y+b1 and x1,y1 = x+a2,y+b2 in  
    let rxm = (float(x1-x0)) /. 2. and rym = (float(y1-y0))  
    let xm = (float x0) +. rxm and ym = (float y0) +. rym in  
    Graphics.set_color col;  
    Graphics.moveto x0 y0;  
    Graphics.lineto x1 y1;  
    let a = atan2 rym rxm in  
    display_arrow (xm,ym) a;  
    display_label (xm,ym) a  
    (string_of_float(float_of_cout g.m.(i).(j))));;  
val display_arc : 'a gg -> Graphics.color -> int -> int
```

Affichage d'un chemin

L'affichage d'un chemin applique le tracé des différents arcs par lesquels passe le chemin. Cet affichage du chemin vers une destination reprend la technique utilisée pour l'affichage texte.

```
# let rec display_shortest_path gg col dest =  
  let g = gg.rg.g in  
  if appartient dest g then  
    let d = index dest g in  
    let rec aux is =  
      if is = gg.etat.source then ()  
      else (  
        let old = gg.etat.chemins.(is) in  
        display_arc gg col old is;  
        aux old )  
      in  
      if not(a_cout gg.etat.distances.(d)) then Printf.printf  
      else aux d;;  
val display_shortest_path : 'a gg -> Graphics.color -> 'a
```

Affichage du graphe

La fonction `display_gg` affiche un graphe complet. Dans le cas où le sommet destination est non vide, elle trace le chemin entre l'origine et la destination.

```
# let display_gg gg () =  
  Upi.display_rect gg.main ();  
  for i=0 to gg.rg.g.ind -1 do  
    for j=0 to gg.rg.g.ind -1 do  
      if i<> j then display_arc gg (Graphics.black) i j  
    done  
  done;  
  if snd gg.dest != Upi.empty_component then  
    display_shortest_path gg Graphics.red (fst gg.dest);;  
val display_gg : 'a gg -> unit -> unit = <fun>
```

Il reste à dessiner les sommets. Comme l'interaction avec l'utilisateur proviendra de la sélection des sommets source et destination, nous définissons un composant pour les sommets.

Composant sommet

La principale action d'un utilisateur est de choisir les extrémités du chemin à rechercher. Pour cela un sommet doit être un composant qui réagit au clic souris et posséder un état lui indiquant si l'on vient de choisir la source ou la destination. On choisit le composant bouton qui effectue une action à la suite d'un clic souris.

Action des sommets

Il est nécessaire de visualiser la sélection d'un sommet. Pour cela on inverse la couleur du fond de celui-ci par la fonction inverse.

```
# let inverse b =  
  let gc = Upi.get_gc b in  
  let fcol = Upi.get_gc_fcol gc  
  and bcol = Upi.get_gc_bcol gc in  
  Upi.set_gc_bcol gc fcol;  
  Upi.set_gc_fcol gc bcol;;  
val inverse : Upi.component -> unit = <fun>
```

La fonction `action_b` effectue cette sélection. Elle est appliquée lors d'un clic souris sur un sommet. Elle prend comme paramètres le sommet associé au bouton ainsi que le graphe pour pouvoir modifier la source ou la destination de la recherche. Quand ces deux sommets sont déterminés, alors elle applique la fonction `dij_rapide` pour trouver le chemin de moindre coût.

```
# let action_clic som gg b bs =  
  let (s1,s) = gg.src  
  and (s2,d) = gg.dest in  
  if s == Upi.empty_component then (  
    gg.src <- (som,b); inverse b )  
  else  
    if d == Upi.empty_component then (  
      inverse b;  
      gg.dest <- (som,b);  
      gg.etat <- dij_rapide s1 gg.rg;
```

```

display_shortest_path gg (Graphics.red) som
)
else (inverse s; inverse d;
gg.dest <- (s2, Upi.empty_component);
gg.src <- som, b; inverse b);;
val action_clic : 'a -> 'a gg -> Upi.component -> 'b ->

```

Création de l'interface

La fonction principale de création de l'interface prend un graphe interfacé et une liste d'options et crée les différents composants puis les associe au graphe. Les paramètres sont le graphe (gg), ses dimensions (gw et gh), la liste d'options du graphes et des sommets (lopt) et la liste d'options pour les bords des sommets (lopt2).

```

# let maj_gg gg gw gh lopt lopt2 =
let gc = Upi.make_default_context () in
  Upi.set_gc gc lopt;
  (* calcul de la taille max des boutons *)
let vs = Array.map gg.to_string gg.rg.g.sommets in
let vsize = Array.map Graphics.text_size vs in
let w = Array.fold_right (fun (x,y) -> max x) vsize 0
and h = Array.fold_right (fun (x,y) -> max y) vsize 0
  (* création de main *)
  gg.main <- Upi.create_panel true gw gh lopt;
  gg.main.Upi.display <- display_gg gg;
  (* et création des boutons *)
let vb_bs =
  Array.map (fun x -> x, Upi.create_button (" "^(gg.to_string
  lopt)
  gg.rg.g.sommets in
let f_act_b = Array.map (fun (x, (b, bs)) ->
let ac = action_clic x gg b
in Upi.set_bs_action bs ac) vb_bs in
let bb =
  Array.map (function (_, (b, _)) -> Upi.create_border b lopt2)
in
  Array.iteri

```



```

(fun i (b) -> let x,y = gg.pos.(i) in
  Upi.add_component gg.main b
  ["PosX",Upi.Iopt (x-w/2);
   "PosY", Upi.Iopt (y-h/2)]) bb;
());;
val maj_gg :
  'a gg ->
  int ->
  int -> (string * Upi.opt_val) list -> (string * Upi.opt_val) list
<fun>

```

Les boutons sont créés automatiquement et sont bien ajoutés à la fenêtre principale.

Test de l'interface

Tout est prêt alors pour la création d'une interface. On utilise un graphe dont les sommets sont des chaînes de caractères pour simplifier les fonctions de conversion. On construit le graphe gg de la manière suivante :

```

# let id x = x;;
# let pos = [| 200, 300; 80, 200 ; 100, 100; 200, 100;
# let gg = cree_gg (create_rech_graphe (test_aho())) po
# maj_gg gg 400 400 ["Background", Upi.Copt (Graphics.
  "Foreground",Upi.Copt Graphics.green]
  [ "Relief", Upi.Sopt "Top";"Border_size", Upi.Iopt 2];;

```

L'appel à *Upi.loop true false gg.main;;* lance la boucle d'interaction de la bibliothèque Upi.

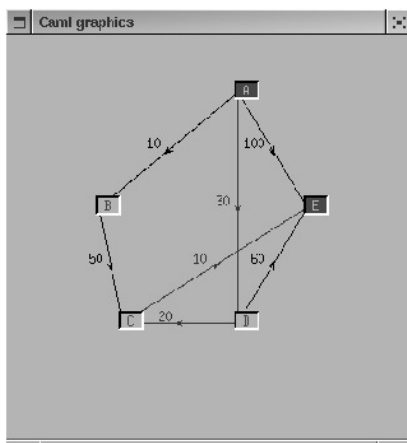


Figure 13.9 : Sélection des sommets d'une recherche

La figure 13.9 montre le chemin trouvé entre le sommet "A" et "E". Les arcs par lesquels passe le chemin changent de couleur.

Application autonome

Nous combinons les différentes étapes rencontrées pour la construction d'une application autonome, c'est-à-dire ne nécessitant pas la présence de la distribution d'Objective CAML sur la machine où elle s'exécute. Cet application prend en argument le nom du fichier de description du graphe.

Fichier de description du graphe

Ce fichier décrit d'une part les informations du graphe ainsi que les informations utiles pour l'interface graphique. Pour celles-ci nous définissons un deuxième format. De cette description graphique, on construit une valeur de type *g_info* suivant.

```
# type g_info = {spos : (int * int) array;
```

```

mutable opt : Upi.lopt;
mutable g_w : int;
mutable g_h : int};;

```

Le format de description des informations graphiques est décrit par les quatre mots clés de la liste key2.

```

# let key2 = ["HAUTEUR"; "LARGEUR"; "POSITION"; "COULEUR"]
val key2 : string list = ["HAUTEUR"; "LARGEUR"; "POSITION"]
# let lex2 l = Genlex.make_lexer key2 (Stream.of_string l)
val lex2 : string -> Genlex.token Stream.t = <fun>

# let pars2 g gi s = match s with parser
| [< '(Genlex.Kwd "HAUTEUR"); '(Genlex.Int i) >] -> gi.g_h <- i
| [< '(Genlex.Kwd "LARGEUR"); '(Genlex.Int i) >] -> gi.g_w <- i
| [< '(Genlex.Kwd "POSITION"); '(Genlex.Ident s);
  '(Genlex.Int i); '(Genlex.Int j) >] -> gi.spos.(index s) <- j
| [< '(Genlex.Kwd "COULEUR"); '(Genlex.Ident s);
  '(Genlex.Int r); '(Genlex.Int g); '(Genlex.Int b) >] ->
  gi.opt <- (s, Upi.Copt (Graphics.rgb r g b))::gi.opt
| [<>] -> ();;
val pars2 : string graphe -> g_info -> Genlex.token Stream.t

```

Création de l'application

La fonction create_gg_graphe prend en entrée un nom de fichier et retourne un couple composé d'un graphe et des informations graphiques associées.

```

# let create_gg_graphe name =
  let g = create_graphe name in
  let gi = {spos = Array.create g.taille (0,0); opt=[]; g_w=0; g_h=0}
  let ic = open_in name in
  try
    print_string ("Loading (pass 2) " ^name ^" : ");
    while true do
      print_string ".";
      let l = input_line ic in pars2 g gi (lex2 l)
    done ;
  g,gi

```



```
-cclib -lgraphics -cclib -L/usr/X11/lib -cclib -lX11
```

La description de la compilation d'un exécutable autonome utilisant la bibliothèque Graphics est donnée aux chapitres [5](#) et [7](#).

Pour en faire plus

Le squelette de cette application est suffisamment généraliste pour pouvoir être réemployé dans un autre cadre que la recherche d'un itinéraire de voyage. En effet différents types de problèmes peuvent se représenter sous forme d'un graphe valué. Par exemple la recherche d'un chemin dans un labyrinthe peut se coder sous forme d'un graphe où chaque carrefour est un sommet du graphe. Trouver une solution revient à déterminer le plus court chemin entre le départ et l'arrivée.

Pour comparer les performances entre C et Objective CAML, on écrit l'algorithme de Dijkstra en C. Le programme C utilisera les structures de données Objective CAML pour effectuer le calcul.

Pour améliorer l'interface graphique, on ajoute un *textfield* pour le nom de fichier et deux boutons pour le chargement ou la sauvegarde d'un graphe. L'utilisateur peut aussi modifier les emplacements des sommets à la souris pour améliorer le rendu.

Une deuxième amélioration de l'interface graphique est de pouvoir choisir la forme des sommets. L'affichage d'un bouton appelle la fonction de tracé d'un rectangle. Rien n'empêche de spécialiser les fonctions d'affichage et d'appartenance par des polygones de description du sommet.

Part III Organisation d'Applications

Présentation de la partie III

La troisième partie de cet ouvrage se consacre à la réalisation d'applications et décrit deux styles d'organisation d'applications : modules et objets. Le but est de modéliser facilement l'application pour un développement incrémentiel et rapide, une maintenance facilitée par une bonne évolutivité et la possibilité de réutiliser de grandes parties pour de futurs développements.

Nous avons déjà présenté les modules prédéfinis (8) du langage vus comme unités de compilation. Le langage de modules d'Objective CAML permet d'une part de définir de nouveaux modules simples, pour construire ses propres bibliothèques, y compris avec des types abstraits, et d'autre part de définir des modules paramétrés par d'autres modules, appelés foncteurs. L'intérêt de ce paramétrage est de pouvoir « appliquer » un module sur différents modules arguments pour créer des modules spécifiques. La communication entre modules est alors explicite via la signature du module paramètre, contenant le type des déclarations globales de celui-ci. Rien n'empêche cependant d'appliquer un foncteur à un module de signature plus importante qui reste néanmoins compatible avec celle indiquée au paramètre.

D'autre part le langage Objective CAML possède une extension objet. La programmation objet permet tout d'abord une modélisation de la communication entre objets. On n'applique plus une fonction à ses arguments, mais on envoie un message (une requête) à un objet qui saura le traiter. L'objet, instance d'une classe (structure de regroupement données/méthodes), exécutera alors le code correspondant. La relation principale entre les classes est l'héritage qui permet de décrire des sous-classes qui possèdent déjà

toutes les déclarations de la classe ancêtre. La liaison retardée entre le nom d'un message et le code correspondant dans l'objet s'effectue à l'exécution du programme. Néanmoins le typage d'Objective CAML garantit qu'un objet receveur possédera toujours une méthode de ce nom, sinon l'inférence de types aura provoqué une erreur à la compilation. La deuxième relation importante est celle de sous-typage, où un objet d'une certaine classe pourra toujours être utilisé à la place d'un objet d'une autre classe. On introduit par là un nouveau type de polymorphisme : le polymorphisme d'inclusion.

Enfin la construction d'interface graphique, abordée au chapitre 5, utilise différents modèles de gestion d'événements. On assemble dans une interface plusieurs composants sur lesquels l'utilisateur ou le système peut produire des événements. L'association d'un composant et du traitement d'un ou plusieurs événements se produisant sur celui-ci permet d'ajouter et de modifier facilement de telles interfaces. L'association composant-événement-traitement peut revêtir plusieurs formes : une fonction (dite callback), ou un héritage avec redéfinition des méthodes de traitement, ou enfin un enregistrement d'un objet traitant (modèle par délégation).

Le chapitre 14 est une présentation de la programmation modulaire. Les différentes terminologies provenant des types abstraits de données et des langages de modules sont exposées et illustrées par des modules simples. Le langage des modules est ensuite détaillé . La correspondance entre modules simples ou non et unités de compilation est explicitée.

Le chapitre 15 contient une introduction à la programmation objet. Elle apporte une nouvelle structuration des programmes Objective CAML, alternative aux modules. Ce chapitre montre comment les notions de programmation objet (héritage simple et multiple, classes abstraites, classes paramétrées, liaison retardée) s'articulent autour du système de types du langage et l'étendent par la relation de sous-typage au polymorphisme d'inclusion.

Le chapitre 16 compare les deux organisations précédentes en précisant les éléments de choix tout en montrant comment simuler l'une par l'autre. Il traite différents cas d'organisations mixtes. Le mélange permet d'enrichir chacune des deux organisations en

particulier avec les classes paramétrées utilisant un type abstrait d'un module.

Le chapitre 17 présente deux applications : des jeux à deux joueurs et la construction d'un monde de robots virtuels. Le premier exemple est organisé en plusieurs modules paramétrés. On y utilise en particulier un module paramétré par la représentation d'un jeu pour l'algorithme minimax ab. Il est ensuite appliqué à deux jeux : Puissance 4 et Stone Henge. Le deuxième exemple utilise une modélisation objet d'un monde et d'un robot abstrait dont on dérive, par héritage, plusieurs simulations. Cet exemple sera repris au chapitre 21.

Programmation modulaire

Introduction

La conception et la programmation modulaire permettent le découpage d'une application en *unités logicielles* et, autant qu'il est possible, le développement séparé de ces unités. Un module est compilable séparément des autres unités constituant l'application. En conséquence, le développeur d'une application utilisant les ressources d'un module n'est pas obligé de disposer des sources du module, seul le code compilé lui suffit pour obtenir l'exécutable final de son application. Cependant, il doit connaître, pour construire le source de sa propre application, les identificateurs (types, fonctions, exceptions, voire modules) fournis par le module. Pour ce faire, un module possède une interface qui décrit les types et les valeurs qu'il fournit.

L'explicitation d'une interface permet de rendre son utilisation indépendante de son implantation. L'utilisateur d'un module ne connaît que les noms des fonctions fournies sans avoir accès à leur implantation. Ainsi, s'il ne change rien à l'ensemble des noms connus, l'administrateur d'un module peut changer du tout au tout son implantation sans que ses utilisateurs aient besoin de s'en apercevoir. Ce qui est un avantage pour la maintenance des applications. L'interface permet, à l'instar de déclarations locales, de masquer certains éléments d'implantation que le concepteur du

module ne souhaite pas publier. La principale application de ce mécanisme est la réalisation de types de données abstraits.

Enfin, les mécanismes de modules évolués (et celui d'Objective CAML en fait partie) permettent la définition de modules paramétrés ou génériques. Ce sont des modules prenant d'autres modules en paramètre ce qui accroît les possibilités de réutilisation de code.

Plan du chapitre

La première section présente et commente la réalisation du module Stack de la distribution d'Objective CAML. On y montre comment obtenir une autre implantation du « même » module. La deuxième section introduit le langage de module d'Objective CAML pour les modules simples et en montre quelques utilisations. En particulier, on étudie comment se pose et se résout le partage de types entre modules. La troisième section introduit la syntaxe et l'utilisation des modules paramétrés que l'on appelle également foncteurs. La quatrième section donne la réalisation de l'exemple classique en programmation modulaire : la gestion d'un compte bancaire avec ses différentes vues (le banquier et le client) et ses différents paramètres (devise, etc.)

Modules simples

Le langage Objective CAML est distribué avec un certain nombre de modules déjà définis. Nous avons vu leur utilisation au chapitre 8. Nous allons voir, dans ce paragraphe, comment réaliser de tels modules.

Interface et implantation

La distribution d'Objective CAML fournit le module Stack implantant les fonctions principales pour utiliser une pile (LIFO).

```
# let queue = Stack.create () ;;  
val queue : '_a Stack.t = <abstr>
```

```
# Stack.push 1 queue ; Stack.push 2 queue ; Stack.push 3
- : unit = ()
# Stack.iter (fun n -> Printf.printf "%d " n) queue ;;
3 2 1 - : unit = ()
```

Objective CAML étant distribué avec ses sources, nous pouvons regarder comment les piles ont été implantées. `ocaml-2.04/stdlib/stack.ml`

```
type 'a t = { mutable c : 'a list } exception Empty let
create () = { c = [] } let clear s = s.c <- [] let push x s = s.c <-
x :: s.c let pop s = match s.c with hd::tl -> s.c <- tl; hd | [] ->
raise Empty let length s = List.length s.c let iter f s = List.iter f s.c
```

Nous nous apercevons que le type des piles connu sous le nom de `Stack.t` est un enregistrement constitué d'un champ modifiable contenant une liste. Les opérations sur la pile sont réalisées par les opérations classiques sur les listes appliquées à l'unique champ de l'enregistrement.

Fort de cette connaissance, rien ne nous interdit a priori d'accéder directement à la liste constituant la queue; cela n'est en fait pas le cas.

```
# let liste = queue.c ;;
```

Characters 13-18:

This expression has type `int Stack.t` but is here used with

Le compilateur proteste comme s'il ne connaissait pas l'identificateur du champ du type `Stack.t`. C'est en fait le cas, comme nous nous en apercevons en regardant l'interface du module `Stack`. `ocaml-2.04/stdlib/stack.mli`

```
(* Module [Stack]: last-in first-out stacks *)
(* This module implements stacks (LIFOs), with in-place modification. *)
type 'a t (* The type of stacks containing elements of type ['a]. *)
exception Empty (* Raised when [pop] is applied to an empty stack. *)
val create: unit -> 'a t (* Return a new stack, initially empty. *)
val push: 'a -> 'a t -> unit (* [push x s] adds the element [x] at the top of stack [s]. *)
val pop: 'a t -> 'a (* [pop s] removes and returns the topmost element in stack [s], or raises [Empty] if the stack is empty. *)
val
```

`clear : 'a t -> unit (* Discard all elements from a stack. *)`
`val length : 'a t -> int (* Return the number of elements in a stack. *)`
`val iter : ('a -> unit) -> 'a t -> unit (* [iter f s] applies [f] in turn to all elements of [s], from the element at the top of the stack to the element at the bottom of the stack. The stack itself is unchanged. *)`
Outre des commentaires expliquant comment utiliser les fonctions du module `Stack`, ce fichier explicite quels sont les valeurs, les types et les exceptions définis dans le fichier `stack.ml`. Plus précisément, l'interface fournit les *noms* des valeurs et leur type. En particulier, le nom du type `t` est donné, mais son implantation (c'est-à-dire l'identificateur du champ `: c`) n'est pas fournie. Ceci explique que l'utilisateur du module `Stack` ne puisse accéder directement à ce champ. Le type `Stack.t` est dit abstrait.

Les fonctions de manipulation des piles sont elles aussi simplement déclarées sans être définies. Il faut seulement, pour que le mécanisme d'inférence de types puisse juger de leur utilisation légitime, indiquer le type de chacune d'elles. Ceci est fait grâce à un nouveau mot clé :

Syntaxe

`val nom : type`

Relation entre interface et implantation

Ainsi le module `Stack` est en fait constitué de deux entités : une interface et une implantation. Pour construire un module, il faut que tous les éléments déclarés dans l'interface soient effectivement définis dans l'implantation. Il est également nécessaire que les définitions des fonctions satisfassent la déclaration de type donnée dans l'interface¹.

La relation entre interface et implantation n'est pas symétrique. L'implantation peut très bien contenir plus de définitions que n'en demande l'interface. Typiquement, la définition d'une fonction un peu complexe pourra utiliser des fonctions auxiliaires dont le nom

n'apparaît pas dans l'interface. Dans ce cas, le programmeur utilisant un tel module, tout comme il ne peut faire référence au champ `c` de la structure `Stack.t`, ne peut faire appel à la fonction auxiliaire non déclarée dans l'interface. De même il est possible dans l'interface d'apporter des restrictions sur le type des valeurs. Imaginons qu'un module définisse la fonction d'identité (`let id x = x`) mais que son interface déclare la valeur `id` de type `int -> nt`; alors les modules utilisant la valeur `id` ne pourront le faire que sur des entiers.

La dualité de constitution d'un module permet de rendre la déclaration des entités composant un module indépendante de leur implantation. Ainsi, on peut substituer au fichier d'implantation `stack.ml` de la distribution, un autre fichier, que l'on appelle toujours `stack.ml` et qui contient une implantation différente des piles reposant, par exemple, sur des tableaux :

```
type 'a t = { mutable sp : int; mutable c : 'a array }
exception Empty
let create () = { sp=0 ; c = [|]| }
let clear s = s.sp <- 0; s.c <- [|]|
let size = 5
let increase s = s.c <- Array.append s.c (Array.create s.c.size 0)

let push x s =
  if s.c = [|]| then ( s.c <- Array.create size x ; s.sp
    else ( if s.sp = Array.length s.c then increase s ) ;
    s.c.(s.sp) <- x ;
    s.sp <- succ s.sp )

let pop s = if s.sp = 0 then raise Empty
  else let x = s.c.(s.sp) in s.sp <- pred s.sp ; x

let length s = Array.length s.c
let iter f s = Array.iter f s.c
```

Cet ensemble de fonctions satisfait le requisit du fichier d'interface `stack.mli`. Le nouveau couple de fichiers `stack.mli` et `stack.ml` (nouvelle mouture) fournit une alternative au module `Stack` de la distribution d'Objective CAML.

Compilation séparée

À l'instar des autres langages de programmation modernes, Objective CAML permet de découper un programme en plusieurs unités de compilation. Une unité de compilation est constituée de deux fichiers, le fichier d'implantation (d'extension `.ml`) et le fichier d'interface (d'extension `.mli`). Chaque unité de compilation est vue comme un module. Ainsi la compilation du fichier `nom.ml` produit le module `Nom`². Les valeurs, types et exceptions définis dans un module peuvent être référencés soit en utilisant la notation pointée (`Module.identificateur`), soit en utilisant le mot clé **open**.

<code>nom1.ml</code>	<code>nom2.ml</code>
type <code>t = { x:int ; y:int } ;;</code>	let val <code>= { Nom1.x = 1 ;</code>
	<code>Nom1.y = 2 } ;;</code>
let <code>f c = c.x + c.y ;;</code>	<code>Nom1.f val ;;</code>
	open <code>Nom1 ;;</code>
	f val <code>;;</code>

Le fichier d'interface, portant l'extension `.mli`, doit être compilé en utilisant la commande `ocamlc -c` avant la compilation des modules qui en dépendent dont en particulier le fichier d'implantation de ce même module. En cas d'absence d'un fichier d'interface, Objective CAML considère que la totalité du module est exportée; c'est-à-dire que toutes les déclarations de l'implantation sont présentes dans l'interface implicite avec leur type le plus général.

L'édition de liens pour engendrer l'exécutable se fait comme décrit au chapitre 7 avec la commande `ocamlc` (sans l'option `-c`) suivi des fichiers objets. Attention, sur la ligne de commande les fichiers doivent être dans l'ordre de leur dépendance; ceci interdit donc les dépendances croisées entre modules.

Pour engendrer un exécutable à partir des fichiers `nom1.ml` et `nom2.ml`, nous utiliserons les commandes suivantes :

```
> ocamlc -c nom1.ml
> ocamlc -c nom2.ml
> ocamlc nom1.cmo nom2.cmo
```

Cette façon de procéder par fichier d'interface et fichier d'implantation pour constituer les modules permet la compilation séparée, mais les possibilités de structuration logique qu'elle offre sont faibles. En particulier, le nom des modules est implicite. Il est déduit du nom des fichiers constituant le module. L'amalgame entre module et fichier empêche d'avoir en même temps deux implantations d'une même interface, voire deux interfaces pour une même implantation. Pour pallier ce défaut, le mécanisme de module a été intégré à la syntaxe du langage de programmation. C'est ce que nous présentons dans la suite de ce chapitre.

Langage des modules simples

Le langage Objective CAML possède un sous-langage de modules qui vient s'ajouter au noyau du langage. Dans ce cadre, l'interface d'un module est appelée sa signature et son implantation est appelée structure. Lorsqu'il n'y a pas d'ambiguïté, nous utiliserons plutôt le terme « module » pour désigner la structure. La syntaxe de déclaration des signatures et des structures est la suivante :

Syntaxe

```
module type NOM =  
sig  
déclarations de l'interface  
end
```

Syntaxe

```
module Nom =  
struct  
définition de l'implantation  
end
```

Warning

Le nom d'un module doit *impérativement* commencer par une

majuscule. Celui d'une signature est libre, mais par convention on utilise des noms en majuscules.

On peut également utiliser les signatures ou des structures anonymes. On écrit alors simplement :

Syntaxe

sig*déclarations***end**

Syntaxe

struct*définitions***end** Nous utiliserons les expressions *signature* et *structure* pour désigner soit des noms de signature et de structure, soit leur expression anonyme.

Toute structure a par défaut une signature calculée par l'inférence de types qui reprend l'intégralité des définitions contenues dans la structure. On peut lors de la définition d'une structure, préciser quelle est la signature attendue en rajoutant une contrainte selon l'une des deux syntaxes suivantes :

Syntaxe

module*Nom:signature* = *structure*

Syntaxe

module*Nom* = (*structure:signature*) Lorsqu'une signature attendue est précisée, le système vérifie que tout ce qui est déclaré dans la signature est défini dans la structure *Nom* et que les types sont cohérents. En d'autres termes, la signature attendue est incluse dans la signature par défaut. Si tel est le cas, *Nom* devient un module de signature *signature* et, de façon analogue à ce qui se passait avec les fichiers d'interface, seules les déclarations apparaissant dans la signature sont accessibles à l'utilisateur du module.

L'accès aux entités déclarées d'un module se fait en utilisant la notation pointée :

Syntaxe

Nom1.nom2 On dit alors que le nom *nom2* est qualifié.

On peut rendre implicite le nom du module en utilisant la directive d'ouverture des modules :

Syntaxe

open*Nom* Dès lors, on peut utiliser les noms des entités sans les qualifier. L'ouverture d'un module provoque, en cas d'identité de nom, le masquage des entités préalablement définies, à la façon des redéfinitions d'identificateurs.

Deux modules pour les piles

Reprenons les piles en utilisant le langage des modules. Nous commençons par définir la signature d'une pile en reprenant les déclarations du fichier `stack.mli` :

```
# module type STACK =  
  sig  
    type 'a t  
    exception Empty  
    val create: unit -> 'a t  
    val push: 'a -> 'a t -> unit  
    val pop: 'a t -> 'a  
    val clear : 'a t -> unit  
    val length: 'a t -> int  
    val iter: ('a -> unit) -> 'a t -> unit  
  end ;;  
module type STACK =  
  sig  
    type 'a t  
    exception Empty
```



```

val create : unit -> 'a t
val push : 'a -> 'a t -> unit
val pop : 'a t -> 'a
val clear : 'a t -> unit
val length : 'a t -> int
val iter : ('a -> unit) -> 'a t -> unit
end

```

On obtient une première implantation des piles en utilisant le module de la bibliothèque standard :

```

# module Stack_distrib = Stack ;;
module Stack_distrib :
  sig
    type 'a t = 'a Stack.t
    exception Empty
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val clear : 'a t -> unit
    val length : 'a t -> int
    val iter : ('a -> unit) -> 'a t -> unit
  end

```

On en définit une seconde utilisant des tableaux :

```

# module Stack_perso =
struct
  type 'a t = { mutable sp : int; mutable c : 'a array }
  exception Empty
  let create () = { sp=0 ; c = [|]| }
  let clear s = s.sp <- 0; s.c <- [|]|
  let size = 5
  let increase s = s.c <- Array.append s.c (Array.create size 0)
  let push x s =
    if s.c = [|]| then ( s.c <- Array.create size x; s.sp <- 0 )
    else ( (if s.sp = Array.length s.c then increase s ) ;
      s.c.(s.sp) <- x ;
      s.sp <- succ s.sp )

```

```

let pop s = if s.sp =0 then raise Empty
else let x = s.c.(s.sp) in s.sp <- pred s.sp ; x
let length s = s.sp
let iter f s = for i=0 to pred s.sp do f s.c.(i) done
end ;;

module Stack_perso :
  sig
    type 'a t = { mutable sp: int; mutable c: 'a array }
    exception Empty
    val create : unit -> 'a t
    val clear : 'a t -> unit
    val size : int
    val increase : 'a t -> unit
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val length : 'a t -> int
    val iter : ('a -> 'b) -> 'a t -> unit
  end

```

Les deux modules utilisent un type concret différent pour implanter le type t.

```

# Stack_distrib.create () ;;
- : '_a Stack_distrib.t = <abstr>
# Stack_perso.create () ;;
- : '_a Stack_perso.t = {Stack_perso.sp=0; Stack_perso.c=...}

```

On retrouve l'abstraction de type en forçant la signature du second module.

```

# module Stack_perso = (Stack_perso : STACK) ;;
module Stack_perso : STACK
# Stack_perso.create() ;;
- : '_a Stack_perso.t = <abstr>

```

Les deux modules Stack_perso et Stack_distrib n'ont en commun que le nom des fonctions qu'ils implantent. Par contre, leurs types sont différents; il n'est donc pas possible d'utiliser les fonctions de l'un pour manipuler les valeurs de l'autre :

```

# let s = Stack_distrib.create() ;;

```

```
val s : '_a Stack_distrib.t = <abstr>
# Stack_perso.push 0 s ;;
Characters 19-20:
This expression has type 'a Stack_distrib.t = 'a Stack.t
but is here used with type int Stack_perso.t
```

Même si les deux modules avaient possédé un type t de même implantation, le fait d'abstraire ce type en contraignant le module avec la signature STACK interdit la possibilité de partager les valeurs entre les deux modules.

```
# module S1 = ( Stack_perso : STACK ) ;;
module S1 : STACK
# module S2 = ( Stack_perso : STACK ) ;;
module S2 : STACK
# let s = S1.create () ;;
val s : '_a S1.t = <abstr>
# S2.push 0 s ;;
Characters 10-11:
This expression has type 'a S1.t but is here used with t
```

Objective CAML ne dispose pour vérifier la compatibilité des types que de leur nom (leur implantation étant abstraite) et ici ils sont différents : $S1.t$ et $S2.t$. C'est précisément cette restriction qui permet l'abstraction de type en interdisant l'accès à la définition des types dont on veut masquer l'implantation.

Modules et portée lexicale

Nous donnons dans ce paragraphe deux exemples d'utilisation de signatures pour masquer certaines déclarations.

Masquage de types

Abstraire un type permet de restreindre ses valeurs à celles qu'il est possible de construire avec les fonctions que déclare la signature du module où ce type est défini. Dans l'exemple suivant, nous obtenons des entiers dont la construction nous assure qu'ils sont obligatoirement différents de 0.

```
# module Int_Star =
  ( struct
    type t = int
    exception Isnul
    let of_int = function 0 -> raise Isnul | n -> n
    let mult = (+)
    end
  :
  sig
    type t
    exception Isnul
    val of_int : int -> t
    val mult : t -> t -> t
    end
  ) ;;

module Int_Star :
  sig type t exception Isnul val of_int : int -> t val mu
```

Masquage de valeurs

Le masquage d'une valeur permet de réaliser un générateur de symboles analogue à celui vu page ??.

On définit la signature GENSYM contenant seulement deux déclarations de fonctions pour la génération de symboles.

```
# module type GENSYM =
  sig
    val reset : unit -> unit
    val next : string -> string
  end ;;
```

On implante ensuite une structure cohérente pour une telle signature :

```
# module Gensym : GENSYM =
  struct
    let c = ref 0
    let reset () = c:=0
```

```

let next s = incr c ; s ^ (string_of_int !c)
end;;
module Gensym : GENSYM

```

La référence `c` de la structure `Gensym` n'est pas accessible en dehors des deux fonctions exportées.

```

# Gensym.reset();;
- : unit = ()
# Gensym.next "T";;
- : string = "T1"
# Gensym.next "X";;
- : string = "X2"
# Gensym.reset();;
- : unit = ()
# Gensym.next "U";;
- : string = "U1"
# Gensym.c;;
Characters 0-8:
Unbound value Gensym.c

```

La déclaration de `c` peut être considérée comme locale à la structure module `Gensym` puisqu'elle est masquée par la signature associée au module. La contrainte de signature nous a permis de reproduire plus simplement la définition des fonctions `reset_s` et `new_s` qui utilisaient une déclaration locale (voir page [??](#)).

Différentes vues d'un même module

Le langage de module avec contraintes de signature permet d'offrir plusieurs vues d'une même structure. On pourra, par exemple avoir un « super-utilisateur » du module `Gensym` qui est capable de remettre à jour le compteur et un utilisateur ordinaire qui ne peut que créer un nouveau symbole sans maîtriser le compteur. Pour obtenir ce dernier, il suffit de poser la signature :

```

# module type USER_GENSYM =
  sig
    val next : string -> string
  end;;

```

On crée ensuite le module correspondant par la déclaration :

```
# module UserGensym = (Gensym : USER_GENSYM) ;;
module UserGensym : USER_GENSYM
# UserGensym.next "U" ;;
- : string = "U2"
# UserGensym.reset() ;;
Characters 0-16:
Unbound value UserGensym.reset
```

Pour réaliser ce nouveau module on a réutilisé le module Gensym. De plus, les deux modules partagent le même compteur :

```
# Gensym.next "U" ;;
- : string = "U3"
# Gensym.reset() ;;
- : unit = ()
# UserGensym.next "V" ;;
- : string = "V1"
```

Partage de types entre modules

L'incompatibilité entre types abstraits signalée un peu avant (page ??) pose problème lorsque l'on désire partager un type abstrait entre plusieurs modules. Nous examinons deux façons de procéder au partage. L'une est une construction explicite du langage de modules, l'autre utilise la structure de bloc lexical des modules.

Partage par contrainte

Illustrons le problème à l'aide du petit exemple suivant. On définit un module *M* qui fournit un type abstrait *M.t*. Nous le restreignons ensuite selon deux signatures différentes n'autorisant pas les mêmes opérations.

```
# module M =
(
```

```

struct
type t = int ref
let create() = ref 0
let add x = incr x
let get x = if !x>0 then (decr x; 1) else failwith "Emp
end
:
sig
type t
val create : unit -> t
val add : t -> unit
val get : t -> int
end
) ;;

# module type S1 =
  sig
    type t
    val create : unit -> t
    val add : t -> unit
  end ;;

# module type S2 =
  sig
    type t
    val get : t -> int
  end ;;

# module M1 = (M:S1) ;;
module M1 : S1
# module M2 = (M:S2) ;;
module M2 : S2

```

Pour obtenir l'identification désirée des types $M1.t$ et $M2.t$, Objective CAML dispose d'une syntaxe pour contraindre un type normalement abstrait dans une signature.

Syntaxe

NOM with type $t_1 = t_2$ and ...

Il s'agit d'une contrainte de type forçant le type t_1 déclaré par la signature *NOM* à être égal au type t_2 .

On peut poser des contraintes globales sur tous les types d'un module en utilisant la contrainte :

Syntaxe

withmodule*Nom1* = *Nom2* En utilisant de telles contraintes de partage, on peut déclarer les deux modules M1 et M2 comme manipulant la même structure de données.

```
# module M1 = (M:S1 with type t = M.t) ;;
module M1 : sig type t = M.t val create : unit -> t val
# module M2 = (M:S2 with type t = M.t) ;;
module M2 : sig type t = M.t val get : t -> int end
# let x = M1.create() in M1.add x ; M2.get x ;;
- : int = 1
```

Partage et sous-modules

Une autre solution pour assurer le partage de type est d'utiliser le mécanisme des sous-modules. En définissant deux sous-modules (M1 et M2) partageant un type de données abstrait d'un module englobant M, nous pouvons parvenir au résultat souhaité.

```
# module M =
( struct
type t = int ref
module M_hide =
struct
let create() = ref 0
let add x = incr x
let get x = if !x>0 then (decr x; 1) else failwith"Empty"
end
module M1 = M_hide
module M2 = M_hide
end
:
```



```

sig
type t
module M1 : sig val create : unit -> t val add : t -> u
module M2 : sig val get : t -> int end
end ) ;;
module M :
  sig
    type t
    module M1 : sig val create : unit -> t val add : t -> u
    module M2 : sig val get : t -> int end
  end

```

On obtient bien le résultat voulu et une valeur créée par M1 peut être manipulée par M2 :

```

# let x = M.M1.create() ;;
val x : M.t = <abstr>
# M.M1.add x ; M.M2.get x ;;
- : int = 1

```

On rajoute cependant un peu de lourdeur par rapport à la solution précédente : l'accès aux fonctions de M1 et M2 se fait via le module englobant M.

Modules simples et extension

Un module est une entité définie une fois pour toutes. En particulier, lorsque nous définissons un type abstrait à l'aide du mécanisme de modules nous ne pouvons plus en étendre les traitements. En particulier, s'il n'a pas été défini de fonction de création, on ne pourra jamais obtenir de valeur de ce type !

Une façon brutale d'augmenter les traitements fournis par un module est d'éditer les sources et de rajouter ce que l'on désire dans la signature et la structure. Mais alors, on n'a plus du tout affaire au même module et toutes les applications qui utilisaient la version originale du module sont à recompiler. Notons cependant que si la redéfinition des composants du module n'a pas modifié les éléments de l'interface originale, il suffit uniquement de recompiler

l'ensemble de l'application sans avoir à modifier ce qui avait été écrit.

Modules paramétrés

Les modules paramétrés sont aux modules ce que les fonctions sont aux valeurs. Tout comme une fonction construit une nouvelle valeur à partir de valeurs paramètres, un module paramétré permet de construire un nouveau module à partir de modules déjà construits. On donne également le nom de foncteurs aux modules paramétrés.

L'introduction des foncteurs dans le langage des modules permet d'accroître les possibilités de réutilisation du code défini dans les structures.

Un foncteur se définit avec une syntaxe fonctionnelle :

Syntaxe

functor(*Nom:signature*) -> structure

```
# module Couple = functor ( Q : sig type t end ) ->
  struct type couple = Q.t * Q.t end ;;
module Couple :
  functor(Q : sig type t end) -> sig type couple = Q.t *
```

Comme pour les fonctions, il existe un raccourci syntaxique à la déclaration de foncteurs :

Syntaxe

module*Nom1*(*Nom2:signature*) = structure

```
# module Couple ( Q : sig type t end ) = struct type cou
module Couple :
  functor(Q : sig type t end) -> sig type couple = Q.t *
```

Un foncteur peut admettre plusieurs paramètres.

Syntaxe

functor (*Nom1 : signature1*) -> :
functor (*Nomn : signaturen*) -> *structure*

On peut, ici également, utiliser le raccourci syntaxique :

Syntaxe

module *Nom* (*Nom1 : signature1*) ... (*Nomn : signaturen*) =
structure

L'application d'un foncteur à ses paramètres se fait selon le schéma syntaxique suivant :

Syntaxe

module*Nom* = *foncteur*(*structure1*) ... (*structuren*) Notez que chaque paramètre est inscrit entre parenthèses. Le résultat de l'application peut être un module simple ou, à nouveau un foncteur, selon le nombre de paramètres réclamés par le foncteur.

Warning

Il n'est pas possible de créer une signature par application d'une signature « fonctorielle » à d'autres signatures.

Un foncteur qui explicite complètement sa communication avec d'autres modules, c'est-à-dire qui ne fait pas référence directement à un élément d'un autre module hormis ses paramètres, est facilement réutilisable. On peut effectivement l'appliquer à différents arguments selon le cadre de son emploi. Il y a un parallèle entre une fonction complètement paramétrée (sans variable libre) et un foncteur complètement paramétré.

Foncteurs et réutilisabilité

La distribution d'Objective CAML fournit trois modules définissant des foncteurs. Deux d'entre eux prennent comme argument un module fournissant un type de données totalement ordonné; c'est-à-dire respectant la signature suivante :

```
# module type OrderedType =  
  sig  
    type t  
    val compare: t -> t -> int  
  end ;;  
module type OrderedType = sig type t val compare : t ->
```

La fonction compare prend deux arguments et rend une valeur négative si le premier est inférieur au second, une valeur nulle s'ils sont égaux, et une valeur positive si le premier est supérieur au second. Voici un exemple de type ordonné.

```
# module OrderedIntPair =  
  struct  
    type t = int * int  
    let compare (x1,x2) (y1,y2) = let q = x1-y1 in if q=0 t  
  end ;;  
module OrderedIntPair :  
  sig type t = int * int val compare : int * int -> int *
```

Le foncteur Make du module Map construit un module qui implante des tables d'association ayant pour clés des valeurs du type ordonné passé en argument. Ce module fournit des services comparables à ceux des listes d'associations du module List mais en utilisant une structure de données légèrement plus complexe (des arbres binaires équilibrés).

```
# module AssocIntPair = Map.Make (OrderedIntPair) ;;  
module AssocIntPair :  
  sig  
    type key = OrderedIntPair.t  
    and 'a t = 'a Map.Make(OrderedIntPair).t  
    val empty : 'a t  
    val add : key -> 'a -> 'a t -> 'a t  
    val find : key -> 'a t -> 'a  
    val remove : key -> 'a t -> 'a t
```

```

val mem : key -> 'a t -> bool
val iter : (key -> 'a -> unit) -> 'a t -> unit
val map : ('a -> 'b) -> 'a t -> 'b t
val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
end

```

Le foncteur `Make` permet de construire des tables d'associations ayant pour clé n'importe quel type de données dont on saura écrire une fonction compare.

Le module `Set` fournit lui aussi un foncteur `Make` prenant en argument un type ordonné et construisant un module implantant les ensembles de valeurs de ce type.

```

# module SetIntPair = Set.Make (OrderedIntPair) ;;
module SetIntPair :
  sig
    type elt = OrderedIntPair.t
    and t = Set.Make(OrderedIntPair).t
    val empty : t
    val is_empty : t -> bool
    val mem : elt -> t -> bool
    val add : elt -> t -> t
    val singleton : elt -> t
    val remove : elt -> t -> t
    val union : t -> t -> t
    val inter : t -> t -> t
    val diff : t -> t -> t
    val compare : t -> t -> int
    val equal : t -> t -> bool
    val subset : t -> t -> bool
    val iter : (elt -> unit) -> t -> unit
    val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
    val cardinal : t -> int
    val elements : t -> elt list
    val min_elt : t -> elt
    val max_elt : t -> elt
    val choose : t -> elt
  end

```

Nous avons avec `SetIntPair.t` un type pour les ensembles de couples

d'entiers avec toutes les fonctions classiques sur les ensembles. L'illustration de la réutilisabilité nous est donné en construisant les ensembles d'ensembles de paires d'entiers.

```
# module SetofSet = Set.Make (SetIntPair) ;;

# let x = SetIntPair.singleton (1,2) ;; (* x =
val x : SetIntPair.t = <abstr>
# let y = SetofSet.singleton SetIntPair.empty ;; (* y =
val y : SetofSet.t = <abstr>
# let z = SetofSet.add x y ;; (* z =
val z : SetofSet.t = <abstr>
```

Le foncteur Make du module Hashtbl fonctionne de manière similaire à celui du module Map mais en construisant des tables de hachage à la place d'arbres équilibrés. L'argument de ce foncteur est un peu différent; outre le type des clés de la table de hachage, il doit fournir simplement une fonction testant l'égalité entre deux clés à la place d'une fonction de comparaison. De plus, il est nécessaire de donner la fonction de hachage, c'est-à-dire une fonction associant un entier à une clé.

```
# module type HashedType =
  sig
    type t
    val equal: t -> t -> bool
    val hash: t -> int
  end ;;

module type HashedType =
  sig type t val equal : t -> t -> bool val hash : t -> int
end

# module IntMod13 =
  struct
    type t = int
    let equal = (=)
    let hash x = x mod 13
  end ;;

module IntMod13 :
  sig type t = int val equal : 'a -> 'a -> bool val hash
end

# module TblInt = Hashtbl.Make (IntMod13) ;;

module TblInt :
```

```

sig
type key = IntMod13.t
and 'a t = 'a Hashtbl.Make(IntMod13).t
val create : int -> 'a t
val clear : 'a t -> unit
val add : 'a t -> key -> 'a -> unit
val remove : 'a t -> key -> unit
val find : 'a t -> key -> 'a
val find_all : 'a t -> key -> 'a list
val mem : 'a t -> key -> bool
val iter : (key -> 'a -> unit) -> 'a t -> unit
end

```

Déclarations locales de modules

Parmi les extensions du langage, il existe la possibilité de déclarer localement un module.

Syntaxe

```

let module Nom = structure
in expr

```

Par exemple, nous souhaitons utiliser le module Set pour écrire une fonction de tri sur les listes d'entiers en insérant chaque élément de la liste dans un ensemble et en récupérant à la fin la liste des éléments de l'ensemble obtenu.

```

# let sort l =
  let module M =
    struct
      type t = int
      let compare = (-)
    end
  in
  let module MSet = Set.Make(M)
  in MSet.elements (List.fold_right MSet.add l MSet.empty)

```

```
val sort : int list -> int list = <fun>
```

```
# sort [ 5 ; 3 ; 8 ; 7 ; 2 ; 6 ; 1 ; 4 ] ;;  
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]
```

Objective CAML ne permet pas de faire sortir d'un module local une valeur ayant un type qui n'est pas connu à l'extérieur de l'expression.

```
# let test =  
  let module Foo =  
    struct  
      type t  
      let id x = (x:t)  
    end  
  in Foo.id ;;
```

Characters 15-101:

This `let module' expression has type Foo.t -> Foo.t

In this type, the locally bound module name Foo escapes

Exemple : gestion de comptes bancaires

Nous terminons ce chapitre par la réalisation d'un petit exemple illustrant les principaux traits de l'utilisation d'une programmation modulaire : abstraction de type ; diverses vues d'un module ; réutilisation de code à l'aide des foncteurs.

Le but de cet exemple est de fournir deux modules de gestion d'un compte bancaire. L'un est destiné au banquier et l'autre au client. Le principe est de réaliser un module paramétré général fournissant toutes les fonctionnalités souhaitables de gestion puis de l'appliquer à certains paramètres en le contraignant avec la signature correspondant à sa destination finale : le banquier ou le client.

Organisation de l'application

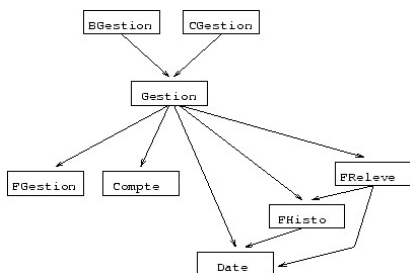


Figure 14.1 : graphe de dépendances des modules

Les deux modules finals BGestion et CGestion sont obtenus par contrainte sur la signature du module Gestion. Ce dernier est obtenu par application du foncteur FGestion aux modules Compte, Date et deux autres modules construits également par application des foncteurs FHisto et FReleve. La figure 14.1 illustre ces dépendances.

Signatures des modules paramètres

Un module de gestion de compte bancaire est paramétré par quatre autres modules dont nous donnons et commentons ci-dessous les signatures.

Le compte lui-même

Ce module fournit essentiellement les opération de calcul sur le contenu du compte.

```

# module type COMPTE = sig
  type t
  exception OperationImpossible
  val creer : float -> float -> t
  val depot : float -> t -> unit
  val retrait : float -> t -> unit
  val solde : t -> float
end

```

```
end ;;
```

Cet ensemble de fonctions fournit les opérations minimales de gestion du contenu du compte. L'opération de création prend en argument le solde initial du compte et la valeur du découvert autorisé. L'opération de retrait peut déclencher l'exception `OperationImpossible`.

Des clés ordonnées

Les opérations seront enregistrées selon un historique décrit au paragraphe suivant. On associera une clé à chaque enregistrement. Les fonctions de gestion des clés sont données par la signature :

```
# module type OCLE =  
  sig  
    type t  
    val creer : unit -> t  
    val of_string : string -> t  
    val to_string : t -> string  
    val eq : t -> t -> bool  
    val lt : t -> t -> bool  
    val gt : t -> t -> bool  
  end ;;
```

La fonction `creer` permet d'engendrer une clé supposée unique. Les fonction `of_string` et `to_string` sont des opérations de conversion de et vers les chaînes de caractères. Les trois autres fonctions sont des fonctions de comparaison.

Historique

Pour garder l'historique de gestion d'un compte, on se donne l'ensemble de données abstraites et de fonctions suivant :

```
# module type HISTO =
```

```

sig
type tcle
type tinfo
type t
val creer : unit -> t
val add : tcle -> tinfo -> t -> unit
val nth : int -> t -> tcle*tinfo
val get : (tcle -> bool) -> t -> (tcle*tinfo) list
end ;;

```

Nous laissons pour l'instant indéterminé la nature des clés d'enregistrement (type *tcle*) la nature des informations (type *tinfo*) et la structure de stockage (type *t*). On supposera que l'ajout de nouvelles informations (fonction *add*) est séquentiel. Le module fournit deux fonctions d'accès aux données archivées : un accès selon leur ordre d'enregistrement (fonction *nth*) et un accès selon un critère défini sur les clés (fonction *get*).

Les relevés de compte

Le dernier paramètre du module de gestion fournit deux fonctions d'édition d'un relevé de compte :

```

# module type RELEVÉ =
  sig
    type tdata
    type tinfo
    val editB : tdata -> tinfo
    val editC : tdata -> tinfo
  end ;;

```

On laisse abstrait le type de données à traiter (*tdata*) ainsi que le type des informations extraites (*tinfo*).

Module général paramétré de gestion

Munis des seules informations que fournissent les signatures ci-

dessus, nous pouvons donner la structure du foncteur général de gestion d'un compte :

```
# module FGestion =  
  functor (C:COMPTE) ->  
  functor (K:OCLE) ->  
  functor (H:HISTO with type tcle=K.t and type tinfo=float) ->  
  functor (R:RELEVÉ with type tdata=H.t and type tinfo=(H.t * float)) ->  
  struct  
    type t = { mutable c : C.t; mutable h : H.t }  
    let creer s d = { c = C.creer s d; h = H.creer() }  
    let depot s g = C.depot s g.c ; H.add (K.creer()) s g.h  
    let retrait s g = C.retrait s g.c ; H.add (K.creer()) s g.h  
    let solde g = C.solde g.c  
    let releve edit g =  
    let f (d,i) = (K.to_string d) ^ ":" ^ (string_of_float i)  
    in List.map f (edit g.h)  
    let releveB = releve R.editB  
    let releveC = releve R.editC  
  end ;;
```

```
module FGestion :  
  functor (C : COMPTE) ->  
  functor (K : OCLE) ->  
  functor  
    (H : sig  
      type tcle = K.t  
      and tinfo = float  
      and t  
      val creer : unit -> t  
      val add : tcle -> tinfo -> t -> unit  
      val nth : int -> t -> tcle * tinfo  
      val get : (tcle -> bool) -> t -> (tcle * tinfo) list  
    end) ->  
  functor  
    (R : sig  
      type tdata = H.t  
      and tinfo = (H.tcle * H.tinfo) list  
      val editB : tdata -> tinfo  
      val editC : tdata -> tinfo  
    end) ->
```

```

sig
type t = { mutable c: C.t; mutable h: H.t }
val creer : float -> float -> t
val depot : H.tinfo -> t -> unit
val retrait : float -> t -> unit
val solde : t -> float
val releve : (H.t -> (K.t * float) list) -> t -> string
val releveB : t -> string list
val releveC : t -> string list
end

```

Précision et partage de types

La contrainte de type appliquée au paramètre H du foncteur FGestion précise que les clés de l'historique sont celles fournies par le paramètre K et que les informations stockées sont simplement des flottants (le montant des transactions). La contrainte de type appliquée au paramètre R indique que les informations traitées par le relevé proviennent de l'historique (paramètre H).

La signature inférée du foncteur FGestion prend en compte les contraintes de type dans la signature inférée des arguments.

On définit le type d'un compte comme le contenu du compte (paramètre C) et son historique.

Opérations

Il est important de remarquer ici que toutes les opérations de ce foncteur sont définies en terme de fonctions fournies par les modules paramètres.

Les opérations de création, retrait et dépôt affectent le contenu du compte et son historique. Les deux opérations de consultation sont la lecture du contenu du compte (fonction solde) et la lecture d'une partie de l'historique (fonction releve).

Définition des paramètres

Il faut, avant de créer les modules finals, définir le contenu des paramètres du foncteur FGestion.

Contenu

Le contenu d'un compte est essentiellement un flottant représentant le solde courant. On rajoute l'information d'autorisation de découvert qui sert à contrôler l'opération de retrait.

```
# module Compte:COMPTE =  
  struct  
    type t = { mutable solde:float; decouvert:float }  
    exception OperationImpossible  
    let creer s d = { solde=s; decouvert=(-.d) }  
    let depot s c = c.solde <- c.solde+.s  
    let solde c = c.solde  
    let retrait s c =  
    let ss = c.solde -. s in  
    if ss < c.decouvert then raise OperationImpossible  
    else c.solde <- ss  
    end ;;  
module Compte : COMPTE
```

Choix d'une clé

On décide que la clé d'enregistrement est simplement la date de la transaction exprimée en flottant telle que la fournit la fonction time du module Unix.

```
# module Date:OCLE =  
  struct  
    type t = float  
    let creer() = Unix.time()  
    let of_string = float_of_string  
    let to_string = string_of_float  
    let eq = (=)  
    let lt = (<)  
    let gt = (>)
```

```

end ;;
module Date : OCLE

```

L'historique

Nous l'avons vu, l'historique d'un compte dépend du type de clé choisi, c'est pourquoi nous définissons un foncteur prenant en argument un module fournissant le type des clés, ce qui permet de définir le type des clés d'enregistrement.

```

# module FHisto (K:OCLE) =
  struct
    type tcle = K.t
    type tinfo = float
    type t = { mutable content : (tcle*tinfo) list }
    let creer() = { content = [] }
    let add c i h = h.content <- (c,i)::h.content
    let nth i h = List.nth h.content i
    let get f h = List.filter (fun (c,_) -> (f c)) h.content
  end ;;
module FHisto :
  functor(K : OCLE) ->
    sig
      type tcle = K.t
      and tinfo = float
      and t = { mutable content: (tcle * tinfo) list }
      val creer : unit -> t
      val add : tcle -> tinfo -> t -> unit
      val nth : int -> t -> tcle * tinfo
      val get : (tcle -> bool) -> t -> (tcle * tinfo) list
    end

```

Remarquons que le type des informations doit être cohérent avec celui donné lors de la définition du foncteur de gestion.

Les relevés

Nous définissons deux types de relevé. Le premier (editB) donne les

cinq dernières transactions et est destiné au banquier ; le second (editC) donne les transactions effectuées les dix derniers jours à partir de la date courante et est destiné au client.

```
# module FReleve (K:OCLE) (H:HISTO with type tcle=K.t) =
  struct
    type tdata = H.t
    type tinfo = (H.tcle*H.tinfo) list
    let editB h =
      List.map (fun i -> H.nth i h) [0;1;2;3;4]
    let editC h =
      let c0 = K.of_string (string_of_float ((Unix.time()) -.
      let f = K.lt c0 in
      H.get f h
    end ;;
module FReleve :
  functor(K : OCLE) ->
  functor
    (H : sig
      type tcle = K.t
      and tinfo
      and t
      val creer : unit -> t
      val add : tcle -> tinfo -> t -> unit
      val nth : int -> t -> tcle * tinfo
      val get : (tcle -> bool) -> t -> (tcle * tinfo) list
    end) ->
  sig
    type tdata = H.t
    and tinfo = (H.tcle * H.tinfo) list
    val editB : H.t -> (H.tcle * H.tinfo) list
    val editC : H.t -> (H.tcle * H.tinfo) list
  end
```

Remarquons que pour définir le relevé décadaire, il nous faut connaître précisément l'implantation des clés utilisées. C'est sans doute là une entorse au principe des types abstraits. Cependant la clé correspondant à une date antérieure de dix jours que nous utilisons est obtenue à partir de sa représentation textuelle par appel à la fonction of_string et non pas par calcul direct de la

représentation interne de cette date (même si notre exemple est trop pauvre pour rendre ce distingo très manifeste).

Les modules finals

Pour construire les modules GBanque et GClient respectivement destinés au banquier et au client nous procédons comme suit :

1. définition d'une structure par application du foncteur FGestion aux paramètres.
2. déclaration d'une signature indiquant les fonctions accessibles dans chacun des cas.
3. définition du module final par contrainte de la structure obtenue en 1 avec la signature déclarée en 2.

```
# module Gestion =
  FGestion (Compte)
  (Date)
  (FHisto(Date))
  (FReleve (Date) (FHisto(Date))) ;;

module Gestion :
  sig
    type t =
      FGestion(Compte) (Date) (FHisto(Date)) (FReleve(Date) (FHisto(Date)))
    { mutable c: Compte.t;
      mutable h: FHisto(Date).t }
    val creer : float -> float -> t
    val depot : FHisto(Date).tinfo -> t -> unit
    val retrait : float -> t -> unit
    val solde : t -> float
    val releve :
      (FHisto(Date).t -> (Date.t * float) list) -> t -> string list
    val releveB : t -> string list
    val releveC : t -> string list
  end

# module type GESTION_BANQUE =
  sig
```

```

type t
val creer : float -> float -> t
val depot : float -> t -> unit
val retrait : float -> t -> unit
val solde : t -> float
val releveB : t -> string list
end ;;

```

```

# module GBanque = (Gestion:GESTION_BANQUE with type t=G
module GBanque :
  sig
    type t = Gestion.t
    val creer : float -> float -> t
    val depot : float -> t -> unit
    val retrait : float -> t -> unit
    val solde : t -> float
    val releveB : t -> string list
  end

```

```

# module type GESTION_CLIENT =
sig
type t
val depot : float -> t -> unit
val retrait : float -> t -> unit
val solde : t -> float
val releveC : t -> string list
end ;;

```

```

# module GClient = (Gestion:GESTION_CLIENT with type t=G
module GClient :
  sig
    type t = Gestion.t
    val depot : float -> t -> unit
    val retrait : float -> t -> unit
    val solde : t -> float
    val releveC : t -> string list
  end

```

Pour que les comptes créés par un banquier puissent être manipulés par un client nous avons utilisé la contrainte de type *Gestion.t* dans la définition des modules GBanque et GClient.

Exercices

Listes d'association

Voici un premier petit exercice simple dans lequel il vous est demandé d'implanter un type abstrait polymorphe : les listes d'association. Et d'en donner deux vues différentes.

1. Définir une signature **ALIST** déclarant un type (abstrait) paramétré par deux variables de type (une pour la clé, l'autre pour la valeur stockée), une fonction de création, une fonction d'ajout, une fonction d'accès, un test d'appartenance et une fonction de suppression. On choisira une implantation fonctionnelle (*ie* : sans effet de bord sur la donnée).

```
# module type ALIST =  
  sig  
    type ('a, 'b) t  
    val create : unit -> ('a, 'b) t  
    val add : 'a -> 'b -> ('a, 'b) t -> ('a, 'b) t  
    val get : 'a -> ('a, 'b) t -> 'b  
    val mem : 'a -> ('a, 'b) t -> bool  
    val rem : 'a -> ('a, 'b) t -> ('a, 'b) t  
  end ;;  
  
module type ALIST =  
  sig  
    type ('a, 'b) t  
    val create : unit -> ('a, 'b) t  
    val add : 'a -> 'b -> ('a, 'b) t -> ('a, 'b) t  
    val get : 'a -> ('a, 'b) t -> 'b  
    val mem : 'a -> ('a, 'b) t -> bool  
    val rem : 'a -> ('a, 'b) t -> ('a, 'b) t
```

end

2. Définir le module **Alist** respectant la signature ALIST

```
# module Alist:ALIST =  
  struct  
    type ('a, 'b) t = ('a*'b) list  
    let create () = []  
    let add k v al = (k,v)::al  
    let get = List.assoc  
    let mem = List.mem_assoc  
    let rem = List.remove_assoc  
  end ;;  
module Alist : ALIST
```

3. Définir une signature **ADM_ALIST** pour l'administrateur des listes d'association. Il pourra uniquement créer une liste, y rajouter et en retirer une entrée.

```
# module type ADM_ALIST =  
  sig  
    type ('a, 'b) t  
    val create : unit -> ('a, 'b) t  
    val add : 'a -> 'b -> ('a, 'b) t -> ('a, 'b) t  
    val rem : 'a -> ('a, 'b) t -> ('a, 'b) t  
  end ;;  
module type ADM_ALIST =  
  sig  
    type ('a, 'b) t  
    val create : unit -> ('a, 'b) t  
    val add : 'a -> 'b -> ('a, 'b) t -> ('a, 'b) t  
    val rem : 'a -> ('a, 'b) t -> ('a, 'b) t  
  end
```

4. Définir une signature **USER_ALIST** pour les utilisateurs des listes qui n'auront droit qu'à l'accès et au test d'appartenance.

```

# module type USER_ALIST =
  sig
    type ('a, 'b) t
    val get : 'a -> ('a, 'b) t -> 'b
    val mem : 'a -> ('a, 'b) t -> bool
  end ;;
module type USER_ALIST =
  sig
    type ('a, 'b) t
    val get : 'a -> ('a, 'b) t -> 'b
    val mem : 'a -> ('a, 'b) t -> bool
  end

```

5. Créer deux **modules** `AdmAlist` et `UserAlist` pour les administrateurs et les utilisateurs. Penser qu'un utilisateur doit pouvoir manipuler une liste créée par un administrateur.

```

# module AdmAlist = (Alist:ADM_ALIST with type ('a,
module AdmAlist :
  sig
    type ('a, 'b) t = ('a, 'b) Alist.t
    val create : unit -> ('a, 'b) t
    val add : 'a -> 'b -> ('a, 'b) t -> ('a, 'b) t
    val rem : 'a -> ('a, 'b) t -> ('a, 'b) t
  end
# module UserAlist = (Alist:USER_ALIST with type ('
module UserAlist :
  sig
    type ('a, 'b) t = ('a, 'b) Alist.t
    val get : 'a -> ('a, 'b) t -> 'b
    val mem : 'a -> ('a, 'b) t -> bool
  end

```

Vecteurs paramétrés

On illustre, dans ce deuxième exercice, comment les modules paramétrés permettent la généricité et la réutilisation de code en définissant un foncteur de manipulation de vecteurs que l'on pourra instancier selon plusieurs types de nombres.

On se donne la signature suivante :

```
# module type NOMBRE =  
  sig  
    type a  
    type t  
    val create : a -> t  
    val add : t -> t -> t  
    val string_of : t -> string  
  end ;;
```

1. Définir le foncteur **FVecteur** paramétré par un module de signature NOMBRE qui définit un type *t*, une fonction de création, l'addition (i.e. la translation) et la conversion en chaîne de caractères.

```
# module FVecteur (N:NOMBRE) =  
  struct  
    type e = N.t  
    type t = { mutable vx:e; mutable vy:e }  
    let create x0 y0 = { vx=x0; vy=y0 }  
    let add v1 v2= {vx = N.add v1.vx v2.vx; vy = N.ad  
    let string_of v= "(" ^N.string_of v.vx^" , "^N.stri  
    end ;;  
  module FVecteur :  
    functor(N : NOMBRE) ->  
    sig  
      type e = N.t  
      and t = { mutable vx: e; mutable vy: e }  
      val create : e -> e -> t  
      val add : t -> t -> t  
      val string_of : t -> string  
    end
```

2. Définir une signature **VECTEUR** , non paramétrée, où les types des nombres et des vecteurs sont abstraits.

```
# module type VECTEUR =
  sig
    type e
    type t
    val create : e -> e -> t
    val add : t -> t -> t
    val string_of : t -> string
  end ;;

module type VECTEUR =
  sig
    type e
    and t
    val create : e -> e -> t
    val add : t -> t -> t
    val string_of : t -> string
  end
```

3. Définir trois **structures** Rationnel, Flottant et Complexe implantant la signature NOMBRE.

```
# module Rationnel:NOMBRE =
  struct
    type a = int*int
    type t = { p:int; q:int }
    let create (p0,q0) = { p=p0; q=q0 }
    let add r1 r2 = { p = r1.p*r2.q + r2.p*r1.q; q = r1.q*r2.p - r2.q*r1.p }
    let string_of r = (string_of_int r.p)^"/"^(string_of_int r.q)
  end ;;

module Rationnel : NOMBRE

# module Flottant:NOMBRE =
  struct
    type a = float
    type t = a
  end
```

```

let create x = x
let add = (+.)
let string_of = string_of_float
end ;;

module Flottant : NOMBRE

# module Complexe:NOMBRE =
  struct
    type a = float*float
    type t = { r:float; i:float }
    let create (r0, i0) = { r=r0; i=i0 }
    let add c1 c2 = { r = c1.r+.c2.r; i = c1.i+.c2.i }
    let string_of c =
      (string_of_float c.r) ^ " + " ^ (string_of_float c.i) ^ "i"
    end ;;
  end

module Complexe : NOMBRE

```

4. Utiliser ces structures pour définir, par application du foncteur FVecteur, trois **modules** pour les rationnels, les réels et les complexes.

```

# module RatVect:VECTEUR = FVecteur(Rationnel) ;;
module RatVect : VECTEUR
# module FloVect:VECTEUR = FVecteur(Flottant) ;;
module FloVect : VECTEUR
# module ComVect:VECTEUR = FVecteur(Complexe) ;;
module ComVect : VECTEUR

```

Arbres lexicaux

Cet exercice reprend les arbres lexicaux vus au chapitre 2, page ??, pour obtenir un module générique de gestion des arbres lexicaux paramétrés par un type abstrait pour les mots.

1. Définir la signature **MOT** contenant la déclaration d'un type abstrait *alpha* pour l'alphabet et *t* pour les mots sur cet

alphabet. On déclarera le mot vide, l'opération de conversion d'un élément de l'alphabet en un mot, l'accès à un élément d'un mot, l'extraction d'un sous-mot, la longueur d'un mot et la concaténation de deux mots.

```
# module type MOT =  
  sig  
    type alpha  
    type t  
    val null : t  
    val of_alpha : alpha -> t  
    val get : t -> int -> alpha  
    val sub : t -> int -> int -> t  
    val length : t -> int  
    val concat : t -> t -> t  
  end ;;
```

```
module type MOT =  
  sig  
    type alpha  
    and t  
    val null : t  
    val of_alpha : alpha -> t  
    val get : t -> int -> alpha  
    val sub : t -> int -> int -> t  
    val length : t -> int  
    val concat : t -> t -> t  
  end
```

2. Définir le foncteur [ArbLex](#), paramétré par un module de signature MOT, qui redéfinit (en fonction des types et opérations sur les mots) le type des arbres lexicaux et les fonctions existe, ajoute et selecte de l'exercice du chapitre [2](#), page [??](#).

```
# module ArbLex (M:MOT) =  
  struct  
    type node = Lettre of M.alpha * bool * t
```

```
and t = node list
```

```
let rec existe m d =
```

```
let aux sm i n =
```

```
match d with
```

```
[] -> false
```

```
| (Lettre (a,b,l))::q when a = M.get sm i ->
```

```
if n = 1 then b else existe (M.sub sm (i+1) (n-1))
```

```
| (Lettre (a,b,l))::q when a = M.get sm i ->
```

```
existe sm q
```

```
in aux m 0 (M.length m)
```

```
let rec ajoute m d =
```

```
let aux sm i n =
```

```
if n = 0 then d
```

```
else
```

```
match d with
```

```
[] ->
```

```
let aux = ajoute (M.sub sm (i+1) (n-1)) [] in
```

```
[ Lettre (M.get sm i, n = 1, aux) ]
```

```
| (Lettre(a,b,l))::q when a = M.get sm i ->
```

```
if n = 1 then (Lettre(a,true,l))::q
```

```
else Lettre(a,b,ajoute (M.sub sm (i+1) (n-1)) l)::
```

```
| (Lettre(a,b,l))::q -> (Lettre(a,b,l))::(ajoute s
```

```
in aux m 0 (M.length m)
```

```
let rec selecte n d = match d with
```

```
[] -> []
```

```
| (Lettre(a,b,l))::q when n=1 ->
```

```
let f (Lettre(a,b,_)) = if b then M.of_alpha a els
```

```
List.filter ((<>) M.null) (List.map f d)
```

```
| (Lettre(a,b,l))::q ->
```

```
let r = selecte (n-1) l and r2 = selecte n q in
```

```
let pr = List.map (function s -> M.concat (M.of_al
```

```
pr@r2
```

```
end ;;
```

Characters 161-409:

Warning: this pattern-matching is not exhaustive.

Here is an example of a value that is not matched:

```
_::_
```

```

module ArbLex :
  functor (M : MOT) ->
  sig
    type node = | Lettre of M.alpha * bool * t
    and t = node list
    val existe : M.t -> t -> bool
    val ajoute : M.t -> t -> t
    val selecte : int -> t -> M.t list
  end

```

3. Définir le module Chars qui implante les opérations de la signature MOT avec $\alpha = \text{char}$ et $t = \text{string}$. En déduire un module **CharsDic** pour les dictionnaires de chaînes de caractères.

```

# module Chars =
  struct
    type alpha = char
    type t = string
    let null = ""
    let of_alpha c = String.make 1 c
    let get s i =
      try s.[i]
      with Invalid_argument(_) -> raise (Invalid_argument _)
    let sub s i1 i2 =
      try String.sub s i1 i2
      with Invalid_argument(_) -> raise (Invalid_argument _)
    let length = String.length
    let concat = (^)
  end ;;
module Chars :
  sig
    type alpha = char
    and t = string
    val null : string
    val of_alpha : char -> string
    val get : string -> int -> char
    val sub : string -> int -> int -> string
    val length : string -> int
  end

```

```

val concat : string -> string -> string
end

# module CharsDic = ArbLex(Chars) ;;
module CharsDic :
  sig
    type node = ArbLex(Chars).node = | Lettre of Chars
    and t = node list
    val existe : Chars.t -> t -> bool
    val ajoute : Chars.t -> t -> t
    val selecte : int -> t -> Chars.t list
  end

```

Résumé

Nous avons fait le tour dans ce chapitre des possibilités qu'offre le langage de modules paramétrés d'Objective CAML.

Comme dans tout mécanisme de modules, on y retrouve la dualité interface-implantation que l'on a renommée dualité signature-structure. Elle permet le masquage de définitions de type, d'exception ou de valeur.

En masquant la représentation des types, on obtient la définition de types abstraits dont les valeurs sont manipulables uniquement avec les moyens fournis dans la signature associée au module. On a vu comment utiliser ce mécanisme pour faciliter le partage de valeurs cachées entre fermetures ainsi que pour offrir plusieurs vues d'une même structure. Dans ce dernier cas, il faut parfois avoir recours à des indications explicites de partage de type.

Le mécanisme plus riche des modules paramétrés (ou foncteurs) nous a permis d'illustrer comment réutiliser le code par un mécanisme aussi simple que l'application d'une fonction.

Pour en savoir plus

Pour avoir plus de détails sur l'implantation des modules dans Objective CAML, on peut consulter le rapport de recherche de

François Pottier qui décrit les motivations et l'implantation du système de modules tel qu'il a été primitivement défini pour le langage Caml-Light. Ce document est un peu technique. Il s'intitule < < Implantation d'un système de modules évolués en Caml-Light > >. Il est accessible à partir de la page du projet Cristal de l'Inria :

Lien: <http://www.inria.fr/Equipes/CRISTAL-fra.html>

Le système de modules d'Objective CAML reprend les principes définis pour le langage SML, cousin d'Objective CAML. La chapitre [22](#) revient sur la comparaison de ces deux langages et donne les pointeurs bibliographiques nécessaires au lecteur curieux.

D'autres langages offrent des systèmes de modules évolués. Citons, par exemple, Modula-3 (et ses différents numéros) ainsi que ADA. Ils offrent la possibilité de définir des modules paramétrés par des types ou des valeurs.

Programmation par objets

Introduction

Comme son nom le laisse supposer, le langage Objective CAML possède la notion d'objet. À la différence de la programmation impérative qui est dirigée par les traitements ou de la programmation fonctionnelle qui est dirigée par les calculs, la programmation par objet est une programmation dirigée par les données. L'utilisation des objets introduit une nouvelle structuration des programmes en classes, les objets étant des instances de celles-ci. Une classe regroupe des données et des traitements. Ces derniers, appelés *méthodes*, définissent les comportements de l'objet. Le déclenchement d'un comportement est effectué par une requête sur l'objet appelée un *envoi de message*. L'objet qui reçoit ce message effectue l'action ou le calcul correspondant à la méthode invoquée dans le message : ce n'est pas une fonction qui est appliquée à des arguments, mais un message (correspondant à un nom de méthode) qui est envoyé à un objet. C'est à l'objet lui-même qu'il revient de déterminer quel est le code effectif à exécuter. Cette liaison *retardée*

entre nom et code permet de faciliter la « modifiabilité » des comportements; c'est-à-dire d'accroître la réutilisabilité du code.

La programmation par objets permet de spécifier les relations entre classes ainsi que la communication entre objets à travers les paramètres passés aux objets dans les messages. Elle offre une nouvelle modélisation des applications par les relations d'agrégation et d'héritage entre classes. Une classe qui hérite d'une autre classe possède toutes les définitions de la classe ancêtre, elle peut étendre les données et les méthodes, et même redéfinir des comportements hérités en respectant le typage. On utilisera une notation graphique¹ pour représenter les relations entre classes.

L'extension objet d'Objective CAML s'intègre dans le système de types du langage. Une déclaration d'une classe définit un type du nom de la classe. La construction d'une instance de cette classe retourne une valeur du type de la classe. Deux genres de polymorphisme coexistent. Le premier est le polymorphisme paramétrique, déjà rencontré avec les types paramétrés, représenté par les classes paramétrées. Le deuxième, appelé polymorphisme d'inclusion, utilise la relation de sous-typage entre objets et la liaison retardée. Si le type de la classe *sc* est sous-type du type de la classe *c*, tout objet de *sc* peut être utilisé à la place d'un objet de *c*. Il faudra cependant que la contrainte entre sous-type et type soit explicite. Le polymorphisme d'inclusion permet de construire des listes hétérogènes où chaque élément est sous-type du type des éléments de la liste. Comme la liaison est retardée, l'envoi du même message à tous les éléments d'une telle liste peut activer des méthodes différentes, propres aux sous-classes.

Par contre Objective CAML n'intègre pas la notion de surcharge des méthodes qui autoriserait différentes définitions d'une même méthode. Cette limitation est induite par l'inférence de types qui tomberait sur des ambiguïtés et demanderait au programmeur d'avoir à fournir de trop nombreuses informations de type supplémentaires.

Il est à noter qu'Objective CAML est le seul langage muni d'une extension objet avec polymorphisme paramétrique et d'inclusion qui reste complètement typé statiquement avec une inférence de types.

Plan du chapitre

Ce chapitre présente l'extension objet du langage Objective CAML. Cette extension ne modifie en rien les éléments du langage déjà étudiés dans les chapitres précédents. Seuls quelques nouveaux mots clés sont réservés pour la syntaxe de la partie objet.

La première section décrit la syntaxe de la déclaration de classe, de l'instanciation d'objet et de l'envoi de message. La deuxième section explique les différentes relations entre classes. La troisième section précise la notion de type objet et montre la richesse de l'extension grâce aux classes abstraites, à l'héritage multiple et à la généricité des classes paramétrées. La quatrième section détaille la relation de sous-typage et montre son intérêt à travers le polymorphisme d'inclusion. La cinquième section s'intéresse au style fonctionnel en objet où l'on ne modifie plus l'état interne d'un objet mais où on retourne une copie modifiée de l'objet receveur. La sixième section précise les autres éléments de l'extension objet que sont les interfaces et déclarations locales dans les classes qui permettent de créer des variables de classe.

Classes, objets et méthodes

L'extension objet d'Objective CAML s'intègre au noyau fonctionnel et au noyau impératif du langage, mais aussi à son système de types. C'est ce dernier point qui en fait son originalité. On obtient ainsi un langage objet, typé statiquement, avec inférence de types. Cette extension permet de définir des classes et des instances, autorise l'héritage entre classes (y compris l'héritage multiple), accepte les classes paramétrées et les classes abstraites. Les interfaces de classes sont engendrées par leur définition mais peuvent être précisées par une signature, à l'instar de ce qui est fait pour les modules.

Terminologie objet

Nous décrivons brièvement dans ce paragraphe les principaux éléments du glossaire de la programmation par objet.

classe

une classe est un ensemble agrégé de champs de données (appelés des variables d'instance) et de traitements (appelés méthodes).

objet

un objet est un élément (ou instance) d'une classe. Un objet possède les comportements de la classe à laquelle il appartient. L'objet est le composant effectif des programmes (c'est lui qui calcule) alors que la classe est plutôt une définition ou une spécification pour l'ensemble des instances à venir.

méthode

une méthode est une action qu'un objet est à même d'effectuer.

envoi de message

un envoi de message à un objet est la demande faite à ce dernier d'exécuter une de ses méthodes. On pourra également dire que l'on invoque une méthode.

Déclaration d'une classe

La syntaxe la plus simple pour définir une classe est la suivante. Nous enrichirons cette définition tout au long du chapitre.

Syntaxe

```
class nom p1 ...pn =  
object  
:  
  variables d'instance  
:  
  méthodes  
:  
end
```

p1, ..., *p_n* sont les paramètres que prendra le constructeur de cette classe. Une classe peut n'avoir aucun paramètre. Une variable

d'instance se déclare par :

Syntaxe

val *nom* = *expr*

ou

val mutable *nom* = *expr*

Dans la mesure où un champ de données est déclaré **mutable**, il est possible d'en modifier sa valeur. Dans le cas contraire, la valeur est celle calculée à la création de l'objet par l'évaluation de *expr*. Une méthode se déclare par :

Syntaxe

method*nom**p1* ...*p_n* = *expr* Il existe d'autres clauses que **val** et **method** dans la déclaration d'une classe. Nous les introduirons au fur et à mesure des besoins.

Une première classe

Commençons par l'inévitable classe point; elle contient :

- deux champs de données *x* et *y* contenant les coordonnées du point,
- six méthodes :
 - deux méthodes d'accès aux champs de données (*get_x* et *get_y*),
 - deux procédures de déplacement absolu (*moveto*) et relatif (*rmoveto*),
 - une méthode de présentation des données sous forme de *string* (*to_string*),
 - une méthode de calcul de la distance du point par rapport à l'origine (*distance*).

```
# class point (x_init,y_init) =  
  object
```

```

val mutable x = x_init
val mutable y = y_init
method get_x = x
method get_y = y
method moveto (a,b) = x <- a ; y <- b
method rmoveto (dx,dy) = x <- x + dx ; y <- y + dy
method to_string () =
  "( " ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^ ")"
method distance () = sqrt (float(x*x + y*y))
end ;;

```

Remarquons que les méthodes peuvent ne pas avoir de paramètre. C'est le cas de `get_x` et de `get_y`. Nous prendrons comme convention de définir les accesseurs aux variables d'instance par des méthodes sans paramètre.

Après la déclaration de la classe `point`, le système affiche le texte suivant :

```

class point :
  int * int ->
  object
  val mutable x : int
  val mutable y : int
  method distance : unit -> float
  method get_x : int
  method get_y : int
  method moveto : int * int -> unit
  method rmoveto : int * int -> unit
  method to_string : unit -> string
end

```

Ce texte contient deux informations. D'une part, un type pour les objets de cette classe. Ce type sera abrégé par le nom *point*. Le type d'un objet est la liste des noms des méthodes de sa classe avec leur type. Dans notre exemple, *point* est une abréviation pour :

```

< distance : unit -> unit; get_x : int; get_y : int;
moveto : int * int -> unit; rmoveto : int * int -> unit

```

```
to_string : unit -> unit >
```

D'autre part, un constructeur d'instances de la classe `point`, de type `int*int -> oint`, qui permet de construire un objet `point` à partir des valeurs initiales fournies en argument. Ici, on construit un point à partir d'un couple d'entiers (sa position initiale). La fonction de construction `point` sera utilisée avec le mot réservé **new**.

Il est possible de définir des types de classe :

```
# type simple_point = < get_x : int; get_y : int; to_string : unit -> string >  
type simple_point = < get_x : int; get_y : int; to_string : unit -> string >
```

Remarque

Le type `point` ne reprend pas la totalité des informations affichées après la déclaration de la classe. Les variables d'instance n'apparaissent pas dans le type. On ne peut accéder à ces valeurs que par l'entremise d'une méthode.

Warning

Une déclaration de classe est considérée comme une déclaration de type. Elle ne peut donc pas contenir de variable de type libre.

Nous reviendrons sur ce point quand nous aborderons les contraintes de type (page ??) et les classes paramétrées (page ??).

Notation graphique des classes

On adapte la notation UML à la syntaxe des types d'Objective CAML. Les classes se notent par un rectangle constitué de trois parties :

- une partie portant le nom de la classe,
- une autre où figurent les attributs (champs de données) d'une instance de la classe,

- une dernière où sont inscrites les méthodes d'une instance de la classe.

La figure 15.1 donne un exemple de représentation graphique de la classe chameau.

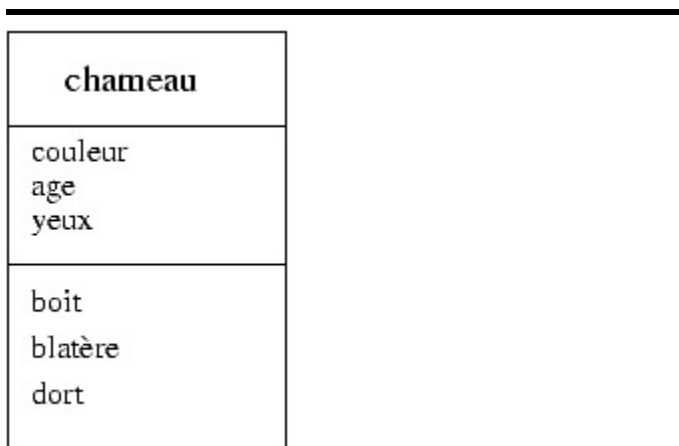


Figure 15.1 : représentation graphique d'une classe

On peut ajouter l'information de types pour les champs d'une classe.

Création d'une instance

Un objet est une valeur d'une classe, appelée instance de cette classe. Cette instance est créée par la primitive de construction générique `new` à laquelle on indique la classe et les valeurs d'initialisation.

Syntaxe

`new nomespr1 ...exprn` L'exemple suivant construit plusieurs instances de la classe `point`, à partir de valeurs initiales différentes.

```
# let p1 = new point (0,0);;
val p1 : point = <obj>
# let p2 = new point (3,4);;
val p2 : point = <obj>
# let coord = (3,0);;
val coord : int * int = 3, 0
# let p3 = new point coord;;
val p3 : point = <obj>
```

En Objective CAML, le constructeur d'une classe est unique mais rien ne nous empêche de déclarer une fonction spécifique `instancie_point` de création de points :

```
# let instancie_point x = new point (x,x) ;;
val instancie_point : int -> point = <fun>
# instancie_point 1 ;;
- : point = <obj>
```

Envoi d'un message

L'envoi d'un message à un objet s'effectue par la notation `#2`.

Syntaxe

nom1#nom2p1 ...pn Le message (la méthode) *nom2* est envoyé à l'objet *nom1*. Les arguments *p1*, ..., *pn* sont ceux attendus par la méthode *nom2*. Elle doit être connue dans la classe de l'objet, c'est-à-dire visible dans son type. Les types des paramètres d'appel doivent satisfaire les types des paramètres formels. L'exemple suivant montre différentes requêtes effectuées sur des objets de la classe `point`.

```
# p1#get_x;;
- : int = 0
# p2#get_y;;
- : int = 4
# p1#to_string();;
```

```

- : string = "( 0, 0) "
# p2#to_string();;
- : string = "( 3, 4) "
# if (p1#distance()) = (p2#distance())
  then print_string ("c'est le hasard\n")
  else print_string ("on pouvait parier\n");;
on pouvait parier
- : unit = ()

```

Du point de vue des types, les objets de type *point* peuvent être manipulés par les fonctions polymorphes d'Objective CAML comme n'importe quelle valeur du langage :

```

# p1 = p1 ;;
- : bool = true
# p1 = p2;;
- : bool = false
# let l = p1::[];;
val l : point list = [<obj>]
# List.hd l;;
- : point = <obj>

```

Warning

L'égalité entre deux objets est vérifiée uniquement dans le cas où ils sont physiquement identiques. Ce point sera détaillé lors de l'étude de la relation de sous-typage (page ??).

Relations entre classes

Les classes d'une application peuvent entretenir entre elles deux types de relations.

1. Une relation d'agrégation, appelée Has-a :

une classe C_2 est dans la relation Has-a avec une classe C_1 si C_2 possède un champ du type de la classe C_1 . Cette relation se généralise en C_2 possède au moins un champ du type de la classe C_1 .

2. Une relation d'héritage, appelée Is-a :

une classe C_2 est sous-classe d'une classe C_1 lorsque C_2 est obtenue par extension du comportement de C_1 . C'est l'avantage majeur de la programmation par objet que de pouvoir étendre le comportement d'une classe existante tout en continuant à utiliser le code écrit pour la classe originale. Quand on étend une classe, la nouvelle classe hérite de tous les champs, de données et de méthodes, de la classe qu'elle étend.

Relation d'agrégation

Une classe C_1 est en relation d'agrégation avec une classe C_2 , si au moins une de ses variables d'instance est du type de la classe C_2 . On indique l'arité de la relation quand elle est connue.

Exemple de relation d'agrégation

On définit une figure comme un ensemble de points. On déclare pour cela la classe *picture* (voir figure 15.2) dont un des champs est un tableau de points. La classe *picture* entretient donc la relation généralisée Has-a avec la classe *point*.

```
# class picture n =  
  object  
    val mutable ind = 0  
    val tab = Array.create n (new point(0,0))  
    method add p =  
      try tab.(ind)<-p ; ind <- ind + 1  
      with Invalid_argument("Array.set")  
    -> failwith ("picture.add:ind =" ^ (string_of_int ind))  
    method remove () = if (ind > 0) then ind <-ind-1  
    method to_string () =  
      let s = ref "["  
      in for i=0 to ind-1 do s:= !s ^ " " ^ tab.(i)#to_string  
         (!s) ^ "]"  
      end ;;
```

```

class picture :
  int ->
  object
  val mutable ind : int
  val tab : point array
  method add : point -> unit
  method remove : unit -> unit
  method to_string : unit -> string
end

```

On construit une figure en créant une instance de la classe *picture* et en ajoutant les points désirés.

```

# let pic = new picture 8;;
val pic : picture = <obj>
# pic#add p1; pic#add p2; pic#add p3;;
- : unit = ()
# pic#to_string ();;
- : string = "[ ( 0, 0) ( 3, 4) ( 3, 0)]"

```

Notation graphique de l'agrégation

La relation entre la classe *picture* et la classe *point* est représentée graphiquement par la figure 15.2. Une flèche, dont l'origine comporte un losange et l'extrémité forme une pointe vide, indique la relation d'agrégation. Dans cet exemple la classe *picture* possède entre 0 et un nombre quelconque de points.

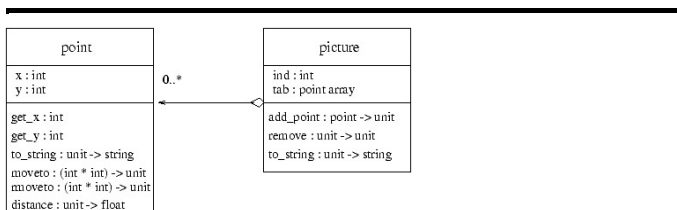


Figure 15.2 : relation d'agrégation

On indique par ailleurs au dessus de la flèche l'arité de la relation.

Relation d'héritage

C'est la relation essentielle de la programmation par objet. Quand une classe *c2* hérite d'une classe *c1* elle récupère tous les champs de la classe ancêtre. Elle peut alors définir de nouveaux champs et même redéfinir des méthodes héritées pour les spécialiser. Comme la classe ancêtre n'a pas été modifiée, les applications s'en servant n'ont pas à être adaptées aux changements apportés à la nouvelle classe.

La syntaxe de la relation d'héritage est la suivante :

Syntaxe

inherit*nom1**p1* ...*p_n* [**as***nom2*]

Les paramètres *p1*, ..., *p_n* sont ceux attendus par le constructeur de la classe *nom1*. Il est possible d'associer un nom à la classe ancêtre pour accéder aux méthodes de celle-ci. On utilise pour cela le mot clé **as**. Cette possibilité est particulièrement utile lorsque la classe fille redéfinit une méthode de la classe ancêtre (voir page ??).

Exemple d'héritage simple

L'exemple classique est d'étendre la classe *point* en ajoutant un attribut de couleur aux points. On définit alors la classe *colored_point* qui hérite de la classe *point*. La couleur est représentée par le champ *c* de type *string*. On ajoute une méthode *get_color* qui retourne la valeur de ce champ. La conversion vers une chaîne de caractères doit alors tenir compte de cet attribut et sera donc redéfinie.

```
# class colored_point (x,y) c =  
  object  
  inherit point (x,y)  
  val mutable c = c
```

```

method get_color = c
method set_color nc = c <- nc
method to_string () = "( " ^ (string_of_int x) ^
", " ^ (string_of_int y) ^ ")" ^
" [" ^ c ^ "]" "
end ;;

class colored_point :
  int * int ->
  string ->
  object
  val mutable c : string
  val mutable x : int
  val mutable y : int
  method distance : unit -> float
  method get_color : string
  method get_x : int
  method get_y : int
  method moveto : int * int -> unit
  method rmoveto : int * int -> unit
  method set_color : string -> unit
  method to_string : unit -> string
end

```

Les valeurs initiales pour la construction d'un *colored_point* sont le couple de coordonnées nécessaire pour la construction d'un *point* et la couleur du point coloré.

L'ensemble des méthodes héritées ou nouvellement définies ou redéfinies correspond à l'ensemble des comportements des instances de la classe.

```

# let pc = new colored_point (2,3) "blanc";;
val pc : colored_point = <obj>
# pc#get_color;;
- : string = "blanc"
# pc#get_x;;
- : int = 2
# pc#to_string();;
- : string = "( 2, 3) [blanc]"
# pc#distance;;

```

```
- : unit -> float = <fun>
```

On dit que la classe *point* est la classe ancêtre de la classe *colored_point* et que celle-ci est la classe fille de celle-là.

Warning

La redéfinition d'une méthode par une classe fille doit respecter le type de la méthode définie dans la classe ancêtre.

Notation graphique de l'héritage

La relation d'héritage entre classes se note par une flèche, dont la pointe est formée par un triangle fermé, allant de la classe fille vers la classe ancêtre. Dans la représentation graphique de l'héritage, on ne fait figurer dans la classe fille que les nouveaux champs, les nouvelles méthodes et les méthodes redéfinies. La figure [15.3](#) illustre la relation entre la classe *colored_point* et son ancêtre *point*.

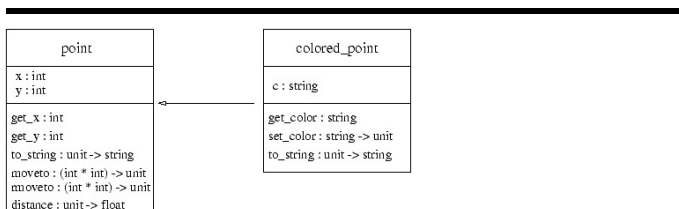


Figure 15.3 : Relation d'héritage

Comme il contient des méthodes supplémentaires, le type *colored_point* est différent du type *point*. Le test d'égalité entre deux instances de ces classes provoque l'affichage d'un long message d'erreur qui rappelle le type complet de chaque classe pour bien montrer leur différence.

```
# p1 = pc;;
```

Characters 6-8:

This expression has type

```
colored_point =
```

```
< distance : unit -> float; get_color : string; get_x :  
int; moveto : int * int -> unit; rmoveto : int * int ->  
set_color : string -> unit; to_string : unit -> string
```

but is here used with type

```
point =
```

```
< distance : unit -> float; get_x : int; get_y : int;  
moveto : int * int -> unit; rmoveto : int * int -> unit  
to_string : unit -> string >
```

Only the first object type has a method `get_color`

Autres traits objet

Référencements :

Il est pratique, dans la définition d'une méthode d'une classe, de pouvoir invoquer une autre méthode de cette même classe ou une méthode de la classe ancêtre. Pour cela Objective CAML autorise à nommer l'objet lui-même ou (les objets de) la classe ancêtre. Dans le premier cas, on indique le nom choisi après le mot clé **object** et dans le second cas, après la déclaration d'héritage.

Par exemple, pour définir la méthode `to_string` des points colorés, il est préférable d'invoquer la méthode `to_string` de la classe ancêtre et d'étendre son comportement en utilisant la nouvelle méthode `get_color`.

```
# class colored_point (x,y) c =  
  object (self)  
    inherit point (x,y) as super  
    val c = c  
    method get_color = c  
    method to_string () = super#to_string() ^ " [" ^ self#  
  end ;;
```

Il est possible de donner des noms quelconques pour la classe ancêtre et sa sous-classe, mais autant utiliser la terminologie objet (self ou this pour soi-même et super pour l'ancêtre). Choisir d'autres noms s'avère cependant utile dans le cas de l'héritage multiple pour différencier les ancêtres (voir page ??).

Warning

On ne peut pas référencer une variable d'instance ancêtre dans le cas où l'on déclare une nouvelle variable d'instance du même nom qui masque la première.

Liaison retardée

On appelle liaison retardée la détermination à l'exécution de la méthode à utiliser lors de l'envoi d'un message. La liaison retardée s'oppose à la liaison statique qui est déterminée à la compilation. En Objective CAML la liaison des méthodes est retardée. C'est donc le receveur d'un message qui sait quel est le code à exécuter.

La déclaration précédente de la classe *colored_point* redéfinit la méthode `to_string`. Cette nouvelle définition utilise la méthode `get_color` de la classe. Définissons à présent une nouvelle classe *colored_point_bis* héritière de *colored_point*, qui redéfinit la méthode `get_color` (en testant la pertinence de la chaîne de caractères) mais qui ne redéfinit pas `to_string`.

```
# class colored_point_bis coord c =  
  object  
    inherit colored_point coord c  
    val true_colors = ["blanc"; "noir"; "rouge"; "vert"; "bleu"]  
    method get_color = if List.mem c true_colors then c else "noir"  
  end ;;
```

La méthode `to_string` est la même dans les deux classes de points colorés, cependant, deux objets de chacune de ces classes n'auront pas le même comportement.

```
# let p1 = new colored_point (1,1) "bleue comme une orange"
```

```

val p1 : colored_point = <obj>
# p1#to_string();;
- : string = "( 1, 1) [bleue comme une orange] "
# let p2 = new colored_point_bis (1,1) "bleue comme une
val p2 : colored_point_bis = <obj>
# p2#to_string();;
- : string = "( 1, 1) [INCONNUE] "

```

La liaison de `get_color` dans le corps de `to_string` n'est donc pas fixée lors de la compilation de la classe *colored_point*. Le code à exécuter lors de l'invocation de la méthode `get_color` est déterminé en fonction des méthodes associées aux instances des classes *colored_point* et *colored_point_bis*. Ainsi, pour une instance de *colored_point*, l'envoi du message `to_string` provoque l'exécution de `get_color` définie dans la classe *colored_point*. Alors que le même envoi de message à une instance de *colored_point_bis* invoque la méthode de la classe ancêtre, cette dernière active la méthode `get_color` de la classe fille qui contrôle la pertinence de la chaîne représentant la couleur.

Représentation mémoire et envoi de messages

Un objet est découpé en deux parties : l'une variable et l'autre fixe. La partie variable contient les variables d'instance, comme pour un enregistrement. La partie fixe correspond à une table des méthodes qui est partagée par toutes les instances de cette classe.

La table des méthodes est un tableau de fonctions et chaque méthode possède un numéro qui est son indice dans ce tableau. On suppose qu'il existe une instruction machine `GETMETHOD(o,n)` qui prend en paramètre un objet `o` et un numéro `n`. Elle retourne la fonction associée à ce numéro dans la table des méthodes. On note `f_n` le résultat de l'appel à `GETMETHOD(o,n)`. La compilation de l'envoi de message `o#m` calcule le numéro `n` de la méthode `m` et engendre le code de l'application `GETMETHOD(o,n)` à l'objet `o`. Il correspond à l'application de la fonction `f_n` à l'objet receveur `o`. La liaison retardée est implantée par l'appel à `GETMETHOD` lors de l'exécution.

L'envoi d'un message à self dans le corps d'une méthode est lui aussi

compilé en une recherche du numéro du message, suivi de l'application de la fonction trouvée dans l'objet lui-même.

Dans le cas d'un héritage, les méthodes, redéfinies ou non, conservent le même numéro. Seule la table change pour les redéfinitions. Ainsi l'envoi du message `to_string` sur une instance de la classe `point` appliquera la fonction de conversion d'un point alors que l'envoi du même message sur une instance de `colored_point` trouvera au même numéro la fonction correspondant à la méthode redéfinie pour tenir compte du champ `couleur`.

C'est cette préservation des numéros qui garantit que le sous-typage (voir page ??) est cohérent du point de vue de l'exécution. En effet si un point coloré est explicitement contraint en point, l'envoi du message `to_string` calcule le numéro de méthode de la classe `point` qui coïncide avec celui de la classe `colored_point`. La recherche de la méthode s'effectuera dans la table associée à l'instance réceptrice, c'est-à-dire la table des `colored_point`.

L'implantation réelle d'Objective CAML est légèrement plus complexe, mais le principe de recherche dynamique de la méthode à employer reste le même.

Initialisation

Il est possible d'indiquer par le mot clé **initializer** dans la définition de la classe, des traitements à effectuer lors de la construction de l'objet. Cette initialisation peut effectuer tous les calculs et les accès aux champs ou aux méthodes de l'instance qui sont normalement autorisés dans une méthode.

Syntaxe

initializer*expr* Reprenons l'exemple de la classe `point` pour définir un point bavard qui annonce sa création.

```
# class verbose_point p =  
  object (self)  
  inherit point p
```

initializer

```
let xm = string_of_int x and ym = string_of_int y
in Printf.printf ">> Création d'un point en (%s %s)\n"
Printf.printf "    a distance %f de l'origine\n" (self#d)
end ;;
```

```
# new verbose_point (1,1);;
>> Création d'un point en (1 1)
    a distance 1.414214 de l'origine
- : verbose_point = <obj>
```

Une utilisation amusante et instructive des initialisateurs est de pouvoir suivre l'héritage des classes lors de la création d'instances. En voici un exemple autant pédagogique que dépouillé.

```
# class c1 =
  object
  initializer print_string "Création d'une instance de c1"
  end ;;
```

```
# class c2 =
  object
  inherit c1
  initializer print_string "Création d'une instance de c2"
  end ;;
```

```
# new c1 ;;
Création d'une instance de c1
- : c1 = <obj>
# new c2 ;;
Création d'une instance de c1
Création d'une instance de c2
- : c2 = <obj>
```

La construction d'une instance de `c2` passe par la construction d'une instance de la classe ancêtre.

Méthodes privées

Une méthode peut être déclarée privée par le mot clé **private**. Elle apparaîtra dans l'interface de la classe mais pas dans le type des instances de cette classe. Une méthode privée peut être invoquée dans la définition des autres méthodes de la classe mais ne peut pas être envoyée à une instance de cette classe. Par contre les méthodes privées sont héritées et pourront donc être utilisées dans les définitions de la hiérarchie³.

Syntaxe

methodprivatenom = *expr* Étendons la classe *point* en lui fournissant une méthode *undo* lui permettant de revenir sur son dernier mouvement. Pour réaliser cela, nous devons nous souvenir de la position occupée avant d'effectuer un mouvement. Nous introduisons donc deux nouveaux champs *old_x* et *old_y* avec leur méthode de mise à jour *mem_pos*. Nous ne voulons pas que l'utilisateur ait accès directement à cette méthode aussi la déclarons nous privée. Il faut redéfinir les méthodes *moveto* et *rmoveto* en mémorisant la position courante avant d'appeler les méthodes initiales de mouvement.

```
# class point_m1 (x0,y0) =
  object(self)
  inherit point (x0,y0) as super
  val mutable old_x = x0
  val mutable old_y = y0
  method private mem_pos () = old_x <- x ; old_y <- y
  method undo () = x <- old_x; y <- old_y
  method moveto (x1, y1) = self#mem_pos () ; super#moveto
  method rmoveto (dx, dy) = self#mem_pos () ; super#rmoveto
  end ;;

class point_m1 :
  int * int ->
  object
  val mutable old_x : int
  val mutable old_y : int
  val mutable x : int
  val mutable y : int
  method distance : unit -> float
```

```

method get_x : int
method get_y : int
method private mem_pos : unit -> unit
method moveto : int * int -> unit
method rmoveto : int * int -> unit
method to_string : unit -> string
method undo : unit -> unit
end

```

Remarquons que la méthode `mem_pos` apparaît dans le type `point_m1` précédée du mot clé **private**, elle ne pourra donc pas être invoquée par une instance de `point` alors que la méthode `undo` le pourra. C'est le même cas de figure que pour les variables d'instance. Si les champs `old_x` et `old_y` figurent dans l'affichage résultant de la compilation ils ne peuvent pas pour autant être manipulés directement (voir [??](#)).

```

# let p = new point_m1 (0, 0) ;;
val p : point_m1 = <obj>
# p#mem_pos() ;;
Characters 0-1:
This expression has type point_m1
It has no method mem_pos
# p#moveto(1, 1) ; p#to_string() ;;
- : string = "( 1, 1)"
# p#undo() ; p#to_string() ;;
- : string = "( 0, 0)"

```

Warning

Une contrainte de type peut rendre publique une méthode déclarée avec l'attribut **private**.

Types et généricité

L'intérêt de la programmation par objet vient d'une part de la modélisation d'un problème par les relations d'agrégation et d'héritage, mais aussi de la possibilité de réutilisation et de modification de comportement des classes. L'extension objet

d'Objective CAML doit de surcroît conserver les propriétés de typage statique du langage. Les classes abstraites permettent de factoriser du code mais aussi de regrouper dans un même « protocole de communication » leurs sous-classes : une classe abstraite fixe les noms et types des messages que pourront recevoir les instances de ses héritières. Cette dernière notion sera encore plus appréciée avec l'héritage multiple.

La notion de type objet ouvert ou, plus simplement, de type ouvert, définissant les méthodes requises, autorise la manipulation d'instances par des fonctions génériques. Cependant, il faudra parfois préciser des contraintes de type. Ce sera indispensable avec les classes paramétrées qui reprennent la généricité du polymorphisme paramétrique dans le cadre des classes. Ce dernier trait de la couche objet d'Objective CAML lui permet d'être vraiment générique.

Classes et méthodes abstraites

Les classes abstraites sont des classes dont certaines méthodes sont déclarées mais ne possèdent pas de corps. Ces méthodes sont alors dites abstraites. Il n'est pas possible d'instancier une classe abstraite : **new** est interdit. On utilise le mot clé **virtual** pour signifier l'abstraction d'une classe ou d'une méthode.

Syntaxe

classvirtualnom = object ...end Une classe doit impérativement être déclarée abstraite dès lors que l'une de ses méthodes l'est. On déclare une méthode abstraite en donnant uniquement son type attendu.

Syntaxe

methodvirtualnom : type Lorsqu'une sous-classe d'une classe abstraite redéfinit *toutes* les méthodes abstraites de son ancêtre, alors elle peut devenir concrète, sinon elle doit elle aussi être déclarée abstraite.

Nous voulons construire un ensemble d'objets affichables disposant tous d'une méthode `print` qui affichera le contenu de l'objet traduit en chaîne de caractères. Ces objets devront donc posséder une méthode `to_string`. Nous définissons à cet effet la classe `printable`. Suivant la nature des objets considérés, la chaîne à construire pourra varier. La méthode `to_string` sera donc abstraite dans la déclaration de `printable`. Du coup, cette classe est également abstraite.

```
# class virtual printable () =
  object (self)
  method virtual to_string : unit -> string
  method print () = print_string (self#to_string())
  end ;;

class virtual printable :
  unit ->
  object
  method print : unit -> unit
  method virtual to_string : unit -> string
  end
```

Notons que le caractère abstrait de la classe et de sa méthode `to_string` est manifeste dans le type obtenu.

À partir de cette classe on cherche à définir la hiérarchie de classes de la figure 15.4.

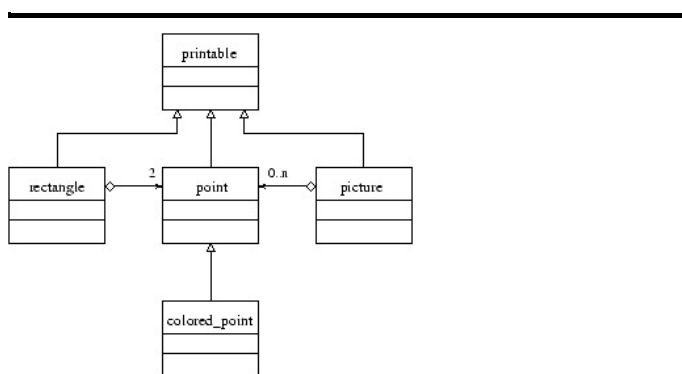


Figure 15.4 : relations entre classes d'objets

affichables

On redéfinit facilement les classes *point*, *colored_point* et *picture*. Il suffit d'ajouter dans leurs déclarations la ligne *inherit printable ()* pour les doter par héritage de la méthode *print*.

```
# let p = new point (1,1) in p#print() ;;
( 1, 1)- : unit = ()
# let pc = new colored_point (2,2) "bleu" in pc#print()
( 2, 2) de couleur bleu- : unit = ()
# let t = new picture 3 in t#add (new point (1,1)) ;
t#add (new point (3,2)) ;
t#add (new point (1,4)) ;
t#print() ;;
[ ( 1, 1) ( 3, 2) ( 1, 4)]- : unit = ()
```

La sous-classe *rectangle* suivante hérite de *printable* et définit la méthode *to_string*. Les variables d'instance *llc* et *ruc* désignent respectivement le point en bas à gauche et le point en haut à droite du rectangle.

```
# class rectangle (p1,p2) =
  object
    inherit printable ()
    val mutable llc = (p1 : point)
    val mutable ruc = (p2 : point)
    method to_string () = "[" ^ p1#to_string() ^ "," ^ p2#to_string() ^ "]"
  end ;;
class rectangle :
  point * point ->
  object
    val mutable llc : point
    val mutable ruc : point
    method print : unit -> unit
    method to_string : unit -> string
  end
```

La classe *rectangle* hérite de la classe abstraite *printable*, et donc récupère la méthode `print`. Elle possède deux variables d'instance *point* : les coins inférieur gauche (`llc`) et supérieur droit (`ruc`). Sa méthode `to_string` envoie le message `to_string` à ses variables d'instance `point`.

```
# let r = new rectangle (new point (2,3), new point (4,5))
val r : rectangle = <obj>
# r#print();;
[( 2, 3), ( 4, 5)]- : unit = ()
```

Classes, types et objets

Rappelons que le type d'un objet est le type de ses méthodes. Par exemple le type *point*, inféré lors de la déclaration de la classe *point*, est une abréviation du type :

```
point =
  < distance : unit -> float; get_x : int; get_y : int;
    moveto : int * int -> unit; rmoveto : int * int -> unit;
    to_string : unit -> string >
```

C'est un type objet fermé. C'est-à-dire que la totalité des méthodes et des types associés qui le composent est déterminée ; il n'y en a pas d'autre. Le mécanisme d'inférence de types, lors d'une déclaration de classe, calcule le type fermé associé à cette classe.

Types ouverts

L'envoi d'un message à un objet fait partie intégrante du langage, on peut donc définir une fonction envoyant un message quelconque à un objet dont le type est non précisé.

```
# let f x = x#get_x ;;
val f : < get_x : 'a; .. > -> 'a = <fun>
```

Le type inféré pour l'argument de `f` est un type objet, puisque l'on

envoie un message à `x`, mais ce type objet est ouvert. Le paramètre `x` de la fonction `f` doit au moins posséder une méthode `get_x`. Comme le résultat de l'envoi de ce message n'est pas utilisé dans la fonction `f`, son type est le plus général possible (c'est-à-dire une variable de type `'a`). Le mécanisme d'inférence de types laisse donc la possibilité d'utiliser la fonction `f` avec n'importe quel objet possédant une méthode `get_x`. L'ouverture du type de `x` est signifiée par les deux points (`..`) à la fin du type `< get_x : 'a; .. >`

```
# f (new point (2,3)) ;;
- : int = 2
# f (new colored_point (2,3) "vert émeraude") ;;
- : int = 2
# class c () =
  object
  method get_x = "J'ai une méthode get_x"
  end ;;
class c : unit -> object method get_x : string end
# f (new c ()) ;;
- : string = "J'ai une m\233thode get_x"
```

L'inférence de types pour les classes peut engendrer des types ouverts, en particulier pour les valeurs initiales de construction des instances. L'exemple suivant construit une classe *couple*, dont les valeurs initiales `a` et `b` possèdent une méthode `to_string`.

```
# class couple (a,b) =
  object
  val p0 = a
  val p1 = b
  method to_string() = p0#to_string() ^ p1#to_string()
  method copy () = new couple (p0,p1)
  end ;;
class couple :
  (< to_string : unit -> string; .. > as 'a) *
  (< to_string : unit -> string; .. > as 'b) ->
  object
  val p0 : 'a
  val p1 : 'b
  method copy : unit -> couple
```

```
method to_string : unit -> string
end
```

Les types de *a* et *b* sont deux types ouverts possédant la méthode `to_string`. On note que ces deux types sont considérés comme différents. Ils sont notés respectivement comme *as 'a* et *as 'b*. Les variables de types *'a* et *'b* sont contraintes par le type ouvert engendré. On désigne le type ouvert construit à partir d'un type fermé *obj_type* en utilisant le symbole dièse :

Syntaxe

#obj_type Le type obtenu contient toutes les méthodes du type *obj_type* et se termine par deux points.

Contrainte de type

Nous avons montré au chapitre sur la programmation fonctionnelle (voir page ??) comment contraindre une expression à posséder un type plus précis que celui engendré par l'inférence. Les types objet, fermés ou ouverts, peuvent être utilisés pour effectuer de telles contraintes. On peut vouloir a priori ouvrir le type d'un objet défini pour lui appliquer une méthode à venir. On utilise alors une contrainte de type objet ouvert :

Syntaxe

(*nom:#type*) Ce qui permet d'écrire :

```
# let g (x : #point) = x#amess;;
val g :
  < amess : 'a; distance : unit -> float; get_x : int; ge
  moveto : int * int -> unit; print : unit -> unit;
  rmoveto : int * int -> unit; to_string : unit -> string
  'a = <fun>
```


La contrainte de type avec *#point* force *x* à avoir au moins toutes les méthodes de *point*, et l'envoi du message *amess* ajoute une méthode au type du paramètre *x*.

Comme dans le reste du langage, l'extension objet d'Objective CAML offre un typage statique obtenu par inférence. Lorsque ce mécanisme ne dispose pas de l'information suffisante pour déterminer le type d'une expression, il lui assigne une variable de type. Nous venons de voir que ce phénomène est aussi valable pour le typage des objets. Il peut engendrer parfois des situations de blocage que l'utilisateur devra résoudre en donnant explicitement des informations de type.

```
# class a_point p0 =  
  object  
    val p = p0  
    method to_string() = p#to_string()  
  end ;;
```

Characters 6-89:

Some type variables are unbound in this type:

```
class a_point :  
  (< to_string : unit -> 'b; .. > as 'a) ->  
  object val p : 'a method to_string : unit -> 'b end  
The method to_string has type unit -> 'a where 'a is unbound
```

On sort de ce blocage en précisant que le paramètre *p0* est de type *#point*.

```
# class a_point (p0 : #point) =  
  object  
    val p = p0  
    method to_string() = p#to_string()  
  end ;;
```

```
class a_point :  
  (#point as 'a) -> object val p : 'a method to_string :
```

Pour éviter de placer les contraintes de type à plusieurs endroits dans la déclaration d'une classe, on utilise la syntaxe suivante :

Syntaxe

constraint*type1* = *type2* L'exemple précédent peut s'écrire en indiquant que le paramètre p0 est de type 'a, puis en posant une contrainte de type sur la variable 'a.

```
# class a_point (p0 : 'a) =  
  object  
    constraint 'a = #point  
    val p = p0  
    method to_string() = p#to_string()  
  end ;;  
class a_point :  
  (#point as 'a) -> object val p : 'a method to_string :
```

Plusieurs contraintes de type peuvent être posées dans une déclaration de classe.

Warning

Un type ouvert ne peut pas apparaître comme type d'une méthode.

Cette contrainte forte provient du fait qu'un type ouvert contient une variable de type non encore instanciée provenant de la suite du type. Comme on ne peut avoir une variable de type libre dans une déclaration de type, une méthode contenant un tel type est rejetée par l'inférence de types.

```
# class b_point p0 =  
  object  
    inherit a_point p0  
    method get = p  
  end ;;
```

Characters 6-77:

Some type variables are unbound in this type:

```
class b_point :  
  (#point as 'a) ->  
  object val p : 'a method get : 'a method to_string : un  
The method get has type #point where .. is unbound
```

Le type de `get` est en fait, à cause de la contrainte **constraint** `'a = #point`, le type ouvert `#point`. Celui-ci contient la variable de type libre dénotée par le double point `(. .)` ce qui n'est pas autorisé.

Héritage et type de `self`

Il existe une exception pour l'apparition d'une variable de type dans le type des méthodes. Il s'agit du cas où une variable désigne le type de l'objet lui-même (`self`). Prenons l'exemple d'une méthode d'égalité qui teste si un point est égal à un autre.

```
# class point_eq (x,y) =  
  object (self : 'a)  
    inherit point (x,y)  
    method eq (p:'a) = (self#get_x = p#get_x) && (self#get_y = p#get_y)  
  end ;;  
class point_eq :  
  int * int ->  
  object ('a)  
    val mutable x : int  
    val mutable y : int  
    method distance : unit -> float  
    method eq : 'a -> bool  
    method get_x : int  
    method get_y : int  
    method moveto : int * int -> unit  
    method print : unit -> unit  
    method rmoveto : int * int -> unit  
    method to_string : unit -> string  
  end
```

Le type de la méthode `eq` est `'a -> bool`, mais la variable de type désigne le type de l'instance lors de sa construction.

On peut hériter de la classe `point_eq` et redéfinir la méthode `eq` dont le type est toujours paramétré par le type de l'instance.

```
# class colored_point_eq (xc,yc) c =  
  object (self : 'a)
```

```

inherit point_eq (xc,yc) as super
val c = (c:string)
method get_c = c
method eq (pc : 'a) = (self#get_x = pc#get_x) && (self
&& (self#get_c = pc#get_c)
end ;;
class colored_point_eq :
  int * int ->
  string ->
  object ('a)
  val c : string
  val mutable x : int
  val mutable y : int
  method distance : unit -> float
  method eq : 'a -> bool
  method get_c : string
  method get_x : int
  method get_y : int
  method moveto : int * int -> unit
  method print : unit -> unit
  method rmoveto : int * int -> unit
  method to_string : unit -> string
end

```

La méthode `eq`, de la classe *colored_point_eq*, est toujours du type `'a -> bool`, mais, cette fois, la variable `'a` désigne le type d'une instance de la classe *colored_point_eq*. La définition de `eq` de la classe *colored_point_eq* masque celle héritée. Les méthodes contenant le type de l'instance dans leur type sont appelées méthodes binaires. Elles créeront des limitations à la relation de sous-typage décrite à la page ??.

Héritage multiple

L'héritage multiple permet d'hériter des champs de données et des méthodes de plusieurs classes. En cas de noms de champs ou de méthodes identiques, seulement la dernière déclaration, dans l'ordre des déclarations d'héritages, sera conservée. Néanmoins, on peut référencer une méthode d'une des classes ancêtres en associant des

noms différents à chaque classe dont on hérite. Cela n'est pas valable pour les variables d'instance. Si une classe héritée masque une variable d'instance d'une classe précédemment héritée, cette dernière variable n'est plus accessible directement. Les différentes classes héritées n'ont pas forcément de lien d'héritage entre elles. L'intérêt de l'héritage multiple est d'augmenter la réutilisabilité des classes.

On définit la classe abstraite *geometric_object* qui déclare deux méthodes virtuelles `compute_area` et `compute_circ` pour le calcul de la surface et du périmètre.

```
# class virtual geometric_object () =  
  object  
    method virtual compute_area : unit -> float  
    method virtual compute_circ : unit -> float  
  end;;
```

On redéfinit alors la classe rectangle de la manière suivante :

```
# class rectangle_bis ((p1,p2) : 'a) =  
  object  
    constraint 'a = point * point  
    inherit printable ()  
    inherit geometric_object ()  
    val mutable llc = p1  
    val mutable ruc = p2  
    method to_string () =  
      "[" ^ p1#to_string() ^ ", " ^ p2#to_string() ^ "]"  
    method compute_area() =  
      float ( abs(ruc#get_x - llc#get_x) * abs(ruc#get_y - llc#get_y) )  
    method compute_circ() =  
      float ( (abs(ruc#get_x - llc#get_x) + abs(ruc#get_y - llc#get_y)) / 2 )  
  end;;
```

```
class rectangle_bis :  
  point * point ->  
  object  
    val mutable llc : point  
    val mutable ruc : point  
    method compute_area : unit -> float  
    method compute_circ : unit -> float
```

```

method print : unit -> unit
method to_string : unit -> string
end

```

Cette implantation de classes respecte le graphe d'héritage de la figure 15.5.

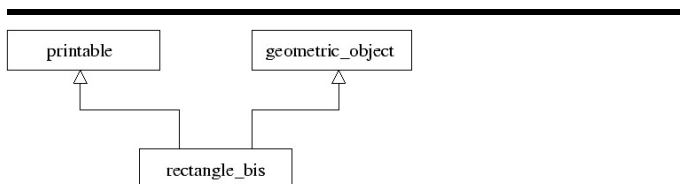


Figure 15.5 : héritage multiple

Pour ne pas avoir à réécrire la méthode de la classe rectangle, on peut hériter directement de rectangle comme le montre le schéma de la figure 15.6.

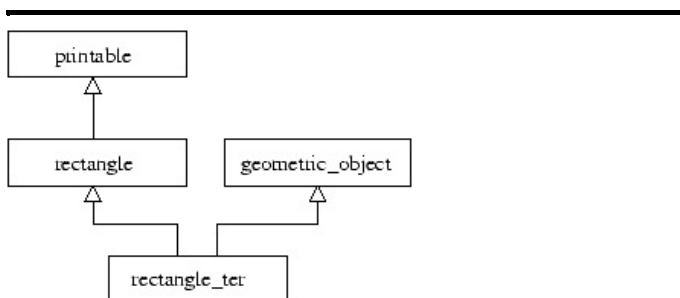


Figure 15.6 : héritage multiple (suite)

Dans ce cas là, seules les méthodes abstraites de la classe abstraite `geometric_object` doivent être définies dans `rectangle_ter`.

```

# class rectangle_ter (p2 : 'a) =

```

```

object
constraint 'a = point * point
inherit rectangle p2
inherit geometric_object ()
method compute_area() =
float ( abs(ruc#get_x - llc#get_x) * abs(ruc#get_y - llc#get_y) )
method compute_circ() =
float ( (abs(ruc#get_x - llc#get_x) + abs(ruc#get_y - llc#get_y)) / 2 )
end;;

```

Toujours dans la même veine, les développements de la hiérarchie `printable` et `geometric_object` auraient pu être séparés jusqu'au moment où il devenait utile d'avoir une classe possédant les deux comportements. Le schéma de la figure 15.7 montre les relations ainsi définies.

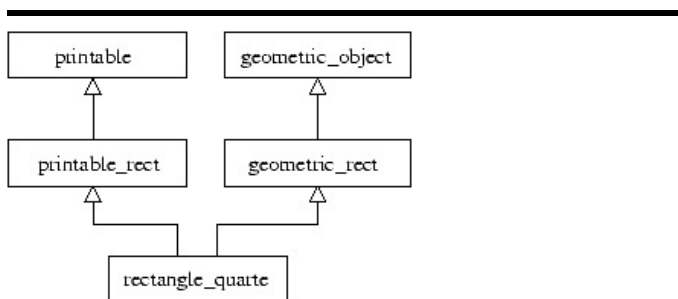


Figure 15.7 : héritage multiple (fin)

Si on suppose que les classes `printable_rect` et `geometric_rect` définissent des variables d'instance pour les coins d'un rectangle, on se retrouve dans la classe `rectangle` avec quatre points (deux par coin).

```

class rectangle_quarte (p1,p2) =
  inherit printable_rect (p1,p2) as super_print
  inherit geometric_rect (p1,p2) as super_geo
end;;

```

Dans le cas où des méthodes de même type existent dans les deux classes ...rect, alors seule la dernière est visible. Néanmoins, grâce au nommage des classes ancêtres (super...), il est toujours possible d'invoquer une méthode de l'une ou l'autre des ancêtres.

L'héritage multiple permet donc une factorisation du code en intégrant les méthodes déjà écrites de différentes classes ancêtres pour construire une nouvelle entité. Le prix à payer est la taille des objets construits plus gros que nécessaire : duplication de champs, héritage de champs inutiles à une application donnée, etc... De plus, en cas de duplication, comme dans notre dernier exemple, il faut établir manuellement la communication entre ces champs (mise à jour, etc...). Dans le dernier exemple sur la classe rectangle, on obtient les variables d'instance de la classe rectangle_graph et rectangle_geo. Si l'une de ces classes possède une méthode qui modifie ces variables (comme un facteur d'échelle) alors il est nécessaire de répercuter ces modifications aux variables héritées de l'autre classe. Cette trop lourde communication entre variables d'instance héritées souligne souvent une mauvaise modélisation du problème posé.

Classes paramétrées

Les classes paramétrées permettent d'utiliser le polymorphisme paramétrique d'Objective CAML dans les classes. Une déclaration de classe peut être paramétrée par des variables de type, comme dans une déclaration de types d'Objective CAML. Cela offre de nouvelles possibilités de généricité, pour la réutilisation du code, et s'intègre dans le typage à la ML quand l'inférence de types produit des types paramétrés.

La syntaxe diffère légèrement de la déclaration de types paramétrés, les paramètres de type sont entre crochets.

Syntaxe

class['a, 'b, ...] *nom* = **object** ...**end** Néanmoins le type Objective CAML est noté comme à l'habitude : ('a,'b,...) *nom*.

Par exemple, si l'on désire construire une classe *pair* pour décrire les paires, une solution naïve consiste à poser :

```
# class pair x0 y0 =  
  object  
    val x = x0  
    val y = y0  
    method fst = x  
    method snd = y  
  end ;;
```

Characters 6-106:

Some type variables are unbound in this type:

```
class pair :
```

```
'a ->
```

```
'b -> object val x : 'a val y : 'b method fst : 'a meth
```

The method `fst` has type `'a` where `'a` is unbound

On retrouve l'erreur de typage évoquée à la définition de la classe *a_point* (page ??). Le message d'erreur indique que la variable de type `'a` assignée au paramètre `x0` (et donc à `x` et `fst`) n'est pas liée.

Pour obtenir un typage correct, comme dans le cas des types paramétrés, il faut paramétrer la classe *pair* avec deux variables de type et forcer le type des paramètres de construction `x0` et `y0` :

```
# class ['a,'b] pair (x0:'a) (y0:'b) =  
  object  
    val x = x0  
    val y = y0  
    method fst = x  
    method snd = y  
  end ;;
```

```
class ['a, 'b] pair :
```

```
'a ->
```

```
'b -> object val x : 'a val y : 'b method fst : 'a meth
```

L'affichage de l'inférence de types indique une interface de classe paramétrée par les variables de type `'a` et `'b`.

Quand on construit une valeur d'une classe paramétrée, les

paramètres de type sont instanciés par les types des paramètres de construction :

```
# let p = new pair 2 'X';;
val p : (int, char) pair = <obj>
# p#fst;;
- : int = 2
# let q = new pair 3.12 true;;
val q : (float, bool) pair = <obj>
# q#snd;;
- : bool = true
```

Remarque

Dans les déclarations de classe, les paramètres de type sont notés entre crochets, alors qu'au niveau de l'affichage des types, ils se retrouvent entre parenthèses.

Héritage de classes paramétrées

L'héritage d'une classe paramétrée doit indiquer les paramètres de cette classe. On définit une classe *acc_pair*, qui hérite de *(a,b) pair* et ajoute deux méthodes d'accès aux champs , get1 et get2,

```
# class ['a,'b] acc_pair (x0 : 'a) (y0 : 'b) =
  object
  inherit ['a,'b] pair x0 y0
  method get1 z = if x = z then y else raise Not_found
  method get2 z = if y = z then x else raise Not_found
  end;;
class ['a, 'b] acc_pair :
  'a ->
  'b ->
  object
  val x : 'a
  val y : 'b
  method fst : 'a
  method get1 : 'a -> 'b
  method get2 : 'b -> 'a
```

```

method snd : 'b
end
# let p = new acc_pair 3 true;;
val p : (int, bool) acc_pair = <obj>
# p#get1 3;;
- : bool = true

```

On peut préciser les paramètres de type de la classe paramétrée héritée, par exemple pour une paire de points.

```

# class pair_point (p1,p2) =
  object
  inherit [point,point] pair p1 p2
  end;;
class pair_point :
  point * point ->
  object
  val x : point
  val y : point
  method fst : point
  method snd : point
  end

```

La classe *pair_point* n'a plus besoin de paramètres de type, car les paramètres 'a et 'b sont complètement déterminés.

Pour construire des paires d'objets affichables, c'est-à-dire possédant une méthode print, on réutilise la classe abstraite printable (voir page ??), puis on définit la classe printable_pair qui hérite de pair.

```

# class printable_pair (x0 ) (y0 ) =
  object
  inherit [printable, printable] acc_pair x0 y0
  method print () = x#print(); y#print ()
  end;;

```

Cette implantation permet effectivement de construire des paires d'instances de printable, mais ne peut pas être utilisée pour des objets d'une autre classe et possédant une méthode print.

Un premier essai consiste à ouvrir le type printable utilisé comme

paramètre de type de acc_pair :

```
# class printable_pair (x0 ) (y0 ) =  
  object  
  inherit [ #printable, #printable ] acc_pair x0 y0  
  method print () = x#print(); y#print ()  
  end;;
```

Characters 6-149:

Some type variables are unbound in this type:

```
class printable_pair :  
  (#printable as 'a) ->  
  (#printable as 'b) ->  
  object  
  val x : 'a  
  val y : 'b  
  method fst : 'a  
  method get1 : 'a -> 'b  
  method get2 : 'b -> 'a  
  method print : unit -> unit  
  method snd : 'b  
  end
```

The method fst has type #printable where .. is unbound

Cet essai échoue car les méthodes fst et snd contiennent un type ouvert.

Nous allons donc conserver les paramètres de type de la classe, tout en les contraignant au type ouvert *#printable*.

```
# class ['a,'b] printable_pair (x0 ) (y0 ) =  
  object  
  constraint 'a = #printable  
  constraint 'b = #printable  
  inherit ['a,'b] acc_pair x0 y0  
  method print () = x#print(); y#print ()  
  end;;  
class ['a, 'b] printable_pair :  
  'a ->  
  'b ->  
  object
```

```

constraint 'a = #printable
constraint 'b = #printable
val x : 'a
val y : 'b
method fst : 'a
method get1 : 'a -> 'b
method get2 : 'b -> 'a
method print : unit -> unit
method snd : 'b
end

```

On construit alors une paire affichable contenant un point et un point coloré.

```

# let pp = new printable_pair
  (new point (1,2)) (new colored_point (3,4) "vert");;
val pp : (point, colored_point) printable_pair = <obj>
# pp#print();;
( 1, 2) ( 3, 4) de couleur vert- : unit = ()

```

Classes paramétrées et typage

Une classe paramétrée est, du point de vue des types, un type paramétré. Une valeur d'un tel type peut alors contenir des variables de type faibles.

```

# let r = new pair [] [];;
val r : ('_a list, '_b list) pair = <obj>
# r#fst;;
- : '_a list = []
# r#fst = [1;2];;
- : bool = false
# r;;
- : (int list, '_a list) pair = <obj>

```

Une classe paramétrée est aussi vue comme un type objet fermé, donc rien n'empêche de l'utiliser aussi comme type ouvert avec la notation dièse.

```

# let compare_rien ( x : ('a, 'a) #pair) =

```

```

if x#fst = x#fst then x#mess else x#mess2;;
val compare_rien : < fst : 'a; mess : 'b; mess2 : 'b; snd : 'b>
<fun>

```

Ce qui nous amène à pouvoir construire des types paramétrés contenant des variables de type faibles, tout en étant des types objet ouverts.

```

# let jolitype x ( y : ('a, 'a) #pair) = if x = y#fst then x#mess else x#mess2;;
val jolitype : 'a -> (('a, 'a) #pair as 'b) -> 'b = <fun>

```

Si on applique cette fonction à un seul paramètre, on obtient une fermeture dont les variables de type deviennent faibles. Un type ouvert, comme *#pair*, contient encore une partie à instancier représentée par les deux points (..). En cela un type ouvert est un paramètre de type dont une partie est déjà connue. Lors de l'affaiblissement d'un tel type suite à une application partielle, l'afficheur précise que la variable de type représentant ce type ouvert est affaiblie. La notation est alors *_#pair*.

```

# let g = jolitype 3;;
val g : ((int, int) _#pair as 'a) -> 'a = <fun>

```

Si maintenant on applique la fonction g à une paire, on modifie son type faible.

```

# g (new acc_pair 2 3);;
- : (int, int) acc_pair = <obj>
# g;;
- : (int, int) acc_pair -> (int, int) acc_pair = <fun>

```

On ne peut plus alors utiliser g sur des paires simples.

```

# g (new pair 1 1);;

```

Characters 4-16:

This expression has type (int, int) pair = < fst : int; snd : int> but is here used with type

```

(int, int) acc_pair =
  < fst : int; get1 : int -> int; get2 : int -> int; snd : int>
Only the second object type has a method get1

```

Enfin comme les paramètres de la classe paramétrée peuvent eux aussi s'affaiblir, on obtient l'exemple suivant.

```
# let h = jolitype [];;
val h : (('b list, 'b list) _#pair as 'a) -> 'a = <fun>
# let h2 = h (new pair [] [1;2]);;
val h2 : (int list, int list) pair = <obj>
# h;;
- : (int list, int list) pair -> (int list, int list) pa
```

Le type du paramètre de `h` n'est plus ouvert. L'application suivante n'est pas typable car l'argument n'est pas de type *pair*.

```
# h (new acc_pair [] [4;5]);;
Characters 4-25:
This expression has type
('a list, int list) acc_pair =
< fst : 'a list; get1 : 'a list -> int list; get2 : int
  snd : int list >
but is here used with type
(int list, int list) pair = < fst : int list; snd : int
Only the first object type has a method get1
```

Remarque

Les classes paramétrées d'Objective CAML sont absolument nécessaires dès que l'on manipule des méthodes dont le type comporte une variable de type autre que le type de `self`.

Sous-typage et polymorphisme d'inclusion

Le sous-typage est la possibilité pour un objet d'un certain type d'être considéré et utilisé comme un objet d'un autre type. Un type d'objet *ot2* pourra être un sous-type de *ot1* si

1. il possède au moins toutes les méthodes de *ot1*,
2. le type de chaque méthode de *ot2* présente dans *ot1* est sous-type de celle de *ot1*.

La relation de sous-typage n'a de sens qu'entre objets. Elle ne devra donc être exprimée qu'entre objets. De plus, la relation de sous-typage devra toujours être explicite. On peut indiquer soit qu'un type est sous-type d'un autre, soit qu'un objet doit être considéré comme objet d'un sur-type.

Syntaxe

```
(nom : sous_type :> sur_type)
(nom :> sur_type)
```

Exemple

Ainsi, on pourra indiquer qu'une instance de `colored_point` peut être utilisée comme une instance de `point` :

```
# let pc = new colored_point (4,5) "blanc";;
val pc : colored_point = <obj>
# let p1 = (pc : colored_point :> point);;
val p1 : point = <obj>
# let p2 = (pc :> point);;
val p2 : point = <obj>
```

Bien que connu comme un point, `p1` n'en reste pas moins un point coloré et l'envoi de la méthode `to_string` déclenchera l'exécution de la méthode attachée aux points colorés :

```
# p1#to_string();;
- : string = "( 4, 5) de couleur blanc"
```

On peut ainsi construire des listes contenant à la fois des points et des points colorés :

```
# let l = [new point (1,2) ; p1] ;;
val l : point list = [<obj>; <obj>]
# List.iter (fun x -> x#print(); print_newline()) l;;
( 1, 2)
```



```
( 4, 5) de couleur blanc  
- : unit = ()
```

Bien entendu, les manipulations que l'on peut faire sur les objets d'une telle liste sont restreintes à celles autorisées sur les points.

```
# p1#get_color () ;;  
Characters 1-3:  
This expression has type point  
It has no method get_color
```

Cette combinaison de liaison tardive et sous-typage autorise une nouvelle forme de polymorphisme : le polymorphisme d'inclusion. C'est-à-dire la possibilité de manipuler des valeurs de n'importe quel type en relation de sous-typage avec le type attendu. Dès lors, l'information de typage statique garantit que l'envoi d'un message trouvera toujours la méthode correspondante, mais le comportement de cette méthode dépendra de l'objet receveur effectif.

Sous-typage n'est pas héritage

Le sous-typage est une notion différente de celle d'héritage. Il y a deux arguments principaux à cela.

Le premier est qu'une instance d'une classe c2 peut avoir comme type un sous-type du type objet c1 sans que c2 soit sous-classe de c1. En effet, on aurait pu définir la classe `colored_point` de manière indépendante de la classe `point` et contraindre le type de l'une de ses instances en type objet `point`.

Le second est qu'il est aussi possible d'avoir un héritage de classes sans pouvoir faire du sous-typage entre instances de ces classes. L'exemple ci-dessous illustre ce second point. Il utilise la possibilité de définir une méthode abstraite prenant en argument une instance (non encore déterminée) de la classe en cours de définition. Dans notre exemple, c'est la méthode `eq` de la classe `equal`.

```
# class virtual equal () =  
  object (self: 'a)
```

```

method virtual eq : 'a -> bool
end;;
class virtual equal : unit -> object ('a) method virtual
# class c1 (x0:int) =
  object (self)
  inherit equal ()
  val x = x0
  method get_x = x
  method eq o = (self#get_x = o#get_x)
  end;;
class c1 :
  int ->
  object ('a) val x : int method eq : 'a -> bool method g
# class c2 (x0:int) (y0:int) =
  object (self)
  inherit equal ()
  inherit c1 x0
  val y = y0
  method get_y = y
  method eq o = (self#get_x = o#get_x) && (self#get_y = o
  end;;
class c2 :
  int ->
  int ->
  object ('a)
  val x : int
  val y : int
  method eq : 'a -> bool
  method get_x : int
  method get_y : int
  end

```

On ne peut pas forcer une instance de c2 à être du type des instances de c1 :

```

# let a = ((new c2 0 0) :> c1) ;;
Characters 11-21:
This expression cannot be coerced to type
  c1 = < eq : c1 -> bool; get_x : int >;
it has type c2 = < eq : c2 -> bool; get_x : int; get_y :

```

```

but is here used with type < eq : c1 -> bool; get_x : int;
Type c2 = < eq : c2 -> bool; get_x : int; get_y : int >
is not compatible with type c1 = < eq : c1 -> bool; get_x : int >
Only the first object type has a method get_y

```

L'incompatibilité entre les types $c1$ et $c2$ vient en fait de ce que le type de `eq` dans $c2$ n'est pas un sous-type du type de `eq` dans $c1$.

Pour montrer qu'il est bon qu'il en soit ainsi, comme dans nos bons vieux devoirs de mathématiques : « *raisonnons par l'absurde* ». Soient $o1$ une instance de $c1$ et $o21$ une instance de $c2$ sous-typée en $c1$. Si nous supposons que le type de `eq` dans $c2$ est un sous-type du type de `eq` dans $c1$ alors l'expression $o21\#eq(o1)$ est correctement typée ($o21$ et $o1$ sont toutes deux de type $c1$). Cependant, à l'exécution, c'est la méthode `eq` de $c2$ qui est déclenchée (puisque $o21$ est une instance de $c2$). Cette méthode va donc tenter d'envoyer le message `get_y` à $o1$ qui ne possède pas une telle méthode !

On aurait donc un système de type qui ne remplirait plus son office. C'est pourquoi la relation de sous-typage entre types fonctionnels doit être définie moins naïvement. C'est ce que nous proposons au paragraphe suivant.

Formalisation

Sous-typage entre objets

Soient $t = \langle m_1:t_1; \dots m_n:t_n \rangle$ et $t' = \langle m_1:s_1; \dots m_n:s_n; m_{n+1}:s_{n+1}; \text{etc} \dots \rangle$ on dit que t' est un sous-type de t , noté $t' \leq t$, si et seulement si $s_i \leq t_i$ pour $i \in \{1, \dots, n\}$.

Appel de fonction

Si $f : t \rightarrow s$, si $a:t'$ et $t' \leq t$ alors (fa) est bien typé et a le type s .

Intuitivement, une fonction f qui attend un argument de type t peut recevoir sans danger un argument d'un sous-type t' de t .

Sous-typage des types fonctionnels

Le type $t' \rightarrow s'$ est un sous type de $t \rightarrow s$, noté $t' \rightarrow s' \leq t \rightarrow s$, si et seulement si

$$s' \leq s \text{ et } t \leq t'$$

La relation $s' \leq s$ est appelée covariance et la relation $t \leq t'$ est appelée contravariance. Cette relation a priori surprenante entre les types fonctionnels peut facilement être justifiée dans le cadre des programmes objets avec liaison dynamique.

Supposons deux classes $c1$ et $c2$ possédant toutes deux une méthode m . La méthode m a le type $t_1 \rightarrow s_1$ dans $c1$ et le type $t_2 \rightarrow s_2$ dans $c2$. Pour plus de lisibilité, notons $m_{(1)}$ la méthode m de $c1$ et $m_{(2)}$ celle de $c2$. Supposons enfin $c2 \leq c1$, c'est à dire $t_2 \rightarrow s_2 \leq t_1 \rightarrow s_1$, et voyons d'où viennent les relations de covariance et de contravariance sur un petit exemple.

Soit $g : s_1 \rightarrow a$, posons $h(o:c1)(x:t_1) = g(o \# m(x))$

covariance

la fonction h attend comme premier argument un objet de type $c1$, comme $c2 \leq c1$ on peut lui passer un objet de type $c2$. La méthode invoquée par $o \# m(x)$ est alors $m_{(2)}$ qui retourne une valeur de type s_2 . Comme cette valeur est passée à g qui attend un argument de type s_1 , il faut bien que $s_2 \leq s_1$.

contravariance

la fonction h attend, comme second argument, une valeur de type t_1 . Si, comme précédemment, nous passons à h un premier argument de type $c2$, la méthode $m_{(2)}$ est invoquée et elle attend un argument de type t_2 . Il faut donc qu'impérativement $t_1 \leq t_2$.

Polymorphisme d'inclusion

On appelle « polymorphisme » la possibilité d'appliquer une fonction à des arguments de n'importe quelle « forme » (type) ou d'envoyer un message à des objets de formes différentes.

Dans le cadre du noyau fonctionnel/impératif du langage, nous avons déjà rencontré le polymorphisme paramétrique qui permet d'appliquer une fonction à des arguments de n'importe quel type. Le paramètre polymorphe de la fonction a un type contenant une variable de type. Une fonction polymorphe est une fonction qui exécutera le même code pour les différents types de paramètres. Pour cela elle n'explore pas la structure de l'argument.

La relation de sous-typage utilisée avec la liaison retardée introduit un nouveau genre de polymorphisme pour les méthodes : le polymorphisme d'inclusion. Celui-ci autorise l'envoi d'un même message, à des instances de types différents, si celles-ci ont été contraintes vers le même sur-type. On construit une liste de points, dont certaines valeurs sont en fait des points colorés (vus comme des points). Le même envoi de message entraîne l'exécution de méthodes différentes, sélectionnées par l'instance réceptrice. Ce polymorphisme est appelé d'inclusion car il accepte l'envoi d'un message, contenu dans la classe c , sur toute instance d'une classe sc , sous-type de c ($sc \leq c$) qui est contrainte en c . On obtient alors un envoi de message polymorphe sur toutes les classes de l'arbre des sous-types de c . À la différence du polymorphisme paramétrique le code exécuté peut être différent pour ces instances.

Les deux formes de polymorphisme peuvent être utilisées conjointement grâce aux classes paramétrées.

Égalité entre objets

Nous pouvons maintenant expliquer le comportement surprenant de l'égalité structurelle entre objets présentée à la page ?? . Un objet est égal structurellement à un autre uniquement s'il est physiquement égal à celui-ci.

```
# let p1 = new point (1,2);;
val p1 : point = <obj>
# p1 = new point (1,2);;
- : bool = false
# p1 = p1;;
- : bool = true
```

Cela provient de la relation de sous-typage. En effet une instance `o2` d'une classe `sc`, sous-type de `c`, contrainte en `c` peut être comparée à une instance `o1` de la classe `c`. Si les champs communs à ces deux instances sont égaux alors les deux objets seraient considérés comme égaux, ce qui est faux du point de vue structurel car `o2` peut avoir des champs supplémentaires. Pour cela Objective CAML considère que deux objets physiquement différents sont structurellement différents.

```
# let pc1 = new colored_point (1,2) "rouge";;
val pc1 : colored_point = <obj>
# let q = (pc1 :> point);;
val q : point = <obj>
# p1 = q;;
- : bool = false
```

C'est une vision restrictive de l'égalité qui garantit qu'une réponse `true` n'est pas erronée; la réponse `false` ne garantit rien.

Style fonctionnel

Le style de la programmation par objets est le plus souvent impératif. Un message est envoyé à un objet qui modifie physiquement son état interne (ses champs de données). Néanmoins il est aussi possible d'aborder la programmation par objets par le style fonctionnel. L'envoi d'un message à un objet retourne un nouvel objet.

Copie d'objets

Objective CAML offre une construction syntaxique particulière qui

permet de retourner une copie de l'objet self dans laquelle les valeurs de certains champs de données sont changés.

Syntaxe

{ <nom1 = expr1;...; nomn = exprn > } On peut ainsi définir des points fonctionnels pour lesquels les méthodes de mouvements relatifs n'ont pas d'effet de bord, mais renvoient un nouveau point.

```
# class f_point p =  
  object  
  inherit point p  
  method f_rmoveto_x (dx) = {< x = x + dx >}  
  method f_rmoveto_y (dy) = {< y = y + dy >}  
  end ;;  
class f_point :  
  int * int ->  
  object ('a)  
  val mutable x : int  
  val mutable y : int  
  method distance : unit -> float  
  method f_rmoveto_x : int -> 'a  
  method f_rmoveto_y : int -> 'a  
  method get_x : int  
  method get_y : int  
  method moveto : int * int -> unit  
  method print : unit -> unit  
  method rmoveto : int * int -> unit  
  method to_string : unit -> string  
  end
```

L'envoi des nouvelles méthodes de déplacement ne modifie pas l'objet receveur.

```
# let p = new f_point (1,1) ;;  
val p : f_point = <obj>  
# print_string (p#to_string()) ;;  
( 1, 1)- : unit = ()  
# let q = p#f_rmoveto_x 2 ;;  
val q : f_point = <obj>
```

```
# print_string (p#to_string()) ;;
( 1, 1)- : unit = ()
# print_string (q#to_string()) ;;
( 3, 1)- : unit = ()
```

Comme ces méthodes construisent un objet, il est possible d'envoyer directement un message au résultat de la méthode `f_rmoveto_x`.

```
# print_string ((p#f_rmoveto_x 3)#to_string()) ;;
( 4, 1)- : unit = ()
```

Les méthodes `f_rmoveto_x` et `f_rmoveto_y` ont un type résultat du type de l'instance de la classe définie. C'est ce qu'indique le `'a` du type de `f_rmoveto_x`.

```
# class f_colored_point (xc, yc) (c:string) =
object
  inherit f_point(xc, yc)
  val color = c
  method get_c = color
end ;;
```

```
class f_colored_point :
  int * int ->
  string ->
  object ('a)
  val color : string
  val mutable x : int
  val mutable y : int
  method distance : unit -> float
  method f_rmoveto_x : int -> 'a
  method f_rmoveto_y : int -> 'a
  method get_c : string
  method get_x : int
  method get_y : int
  method moveto : int * int -> unit
  method print : unit -> unit
  method rmoveto : int * int -> unit
  method to_string : unit -> string
end
```

L'envoi de `f_rmoveto_x` à une instance de `f_colored_point` retourne

une nouvelle instance de `f_colored_point`.

```
# let fpc = new f_colored_point (2,3) "bleu" ;;
val fpc : f_colored_point = <obj>
# let fpc2 = fpc#f_rmoveto_x 4 ;;
val fpc2 : f_colored_point = <obj>
# fpc2#get_c;;
- : string = "bleu"
```

On peut également obtenir une copie de n'importe quel objet en utilisant la primitive `copy` du module `Oo` :

```
# Oo.copy ;;
- : (< .. > as 'a) -> 'a = <fun>
# let q = Oo.copy p ;;
val q : f_point = <obj>
# print_string (p#to_string()) ;;
( 1, 1)- : unit = ()
# print_string (q#to_string()) ;;
( 1, 1)- : unit = ()
# p#moveto(4,5) ;;
- : unit = ()
# print_string (p#to_string()) ;;
( 4, 5)- : unit = ()
# print_string (q#to_string()) ;;
( 1, 1)- : unit = ()
```

Exemple : classe pour les listes

Une méthode fonctionnelle peut utiliser l'objet lui-même `self` pour le calcul de sa valeur de retour. Nous illustrons ce point en définissant une hiérarchie simple de classes pour représenter les listes d'entiers.

On commence par définir la classe abstraite paramétrée par le type des éléments des listes.

```
# class virtual ['a] o_list () =
  object
    method virtual empty : unit -> bool
    method virtual cons : 'a -> 'a o_list
```

```

method virtual head : 'a
method virtual tail : 'a o_list
end;;

```

On définit la classe des listes non vides.

```

# class ['a] o_cons (n ,l) =
  object (self)
    inherit ['a] o_list ()
    val car = n
    val cdr = l
    method empty () = false
    method cons x = new o_cons (x, (self : 'a #o_list :> 'a
    method head = car
    method tail = cdr
    end;;
class ['a] o_cons :
  'a * 'a o_list ->
  object
    val car : 'a
    val cdr : 'a o_list
    method cons : 'a -> 'a o_list
    method empty : unit -> bool
    method head : 'a
    method tail : 'a o_list
  end

```

Il est à noter que la méthode `cons` retourne une nouvelle instance de `'a o_cons`. Pour ce faire le type de `self` est contraint à `'a #o_list` puis sous-typé en `'a o_list`. Sans le sous-typage, on obtient un type ouvert (`'a #o_list`) qui apparaît dans le type des méthodes, ce qui est rigoureusement interdit (voir page ??). Sans la contrainte ajoutée le type de `self` ne peut pas être sous-type de `'a o_list`.

On obtient de cette façon le type attendu pour la méthode `cons`. Et maintenant que l'on connaît le principe, on définit la classe des listes vides.

```

# exception EmptyList ;;
# class ['a] o_nil () =
  object (self)

```

```

inherit ['a] o_list ()
method empty () = true
method cons x = new o_cons (x, (self : 'a #o_list :> 'a
method head = raise EmptyList
method tail = raise EmptyList
end ;;

```

On construit tout d'abord une instance de la liste vide, pour ensuite créer une liste d'entiers.

```

# let i = new o_nil ();;
val i : '_a o_nil = <obj>
# let l = new o_cons (3,i);;
val l : int o_list = <obj>
# l#head;;
- : int = 3
# l#tail#empty();;
- : bool = true

```

La dernière expression envoie le message *tail* sur la liste contenant l'entier 3, qui déclenche la méthode *tail* de la classe *'a o_cons*. Sur ce résultat (la liste vide *i*) est ensuite envoyé le message *empty()* qui retourne *true*. C'est bien la méthode *empty* de la classe *'a o_nil* qui est exécutée.

Autres éléments de l'extension objet

On présente dans cette section la déclaration de types *< >* et surtout les déclarations locales dans les classes. Ces dernières peuvent construire des constructeurs qui possèdent un environnement local, considéré comme un ensemble de variables de classe.

Interface

Les interfaces de classes sont en général inférées par le système d'inférence de types, mais elles peuvent aussi être définies par une déclaration de type. Seules les méthodes publiques apparaissent

dans ce type.

Syntaxe

```
class type nom =  
object  
:  
val nomi : typei  
:  
method nomj : typej  
:  
end
```

On peut ainsi définir l'interface de la classe point :

```
# class type interf_point =  
  object  
    method get_x : int  
    method get_y : int  
    method moveto : (int * int) -> unit  
    method rmoveto : (int * int) -> unit  
    method to_string : unit -> string  
    method distance : unit -> float  
  end ;;
```

L'intérêt de cette déclaration est de pouvoir utiliser le type défini pour une contrainte de type.

```
# let seg_length (p1:interf_point) (p2:interf_point) =  
  let x = float_of_int (p2#get_x - p1#get_x)  
  and y = float_of_int (p2#get_y - p1#get_y) in  
  sqrt ((x*.x) +. (y*.y)) ;;  
val seg_length : interf_point -> interf_point -> float =
```

Les interfaces ne peuvent masquer que les champs de variable d'instance et les méthodes privées. Elles ne peuvent en aucun cas masquer des méthodes abstraites ou des méthodes publiques.

C'est une limitation de leur usage comme le montre l'exemple suivant :

```
# let p = ( new point_m1 (2,3) : interf_point);;
```

Characters 11-29:

This expression has type

```
point_m1 =
```

```
< distance : unit -> float; get_x : int; get_y : int;  
moveto : int * int -> unit; rmoveto : int * int -> unit  
to_string : unit -> string; undo : unit -> unit >
```

but is here used with type

```
interf_point =
```

```
< distance : unit -> float; get_x : int; get_y : int;  
moveto : int * int -> unit; rmoveto : int * int -> unit  
to_string : unit -> string >
```

Only the first object type has a method undo

Néanmoins, les interfaces autorisent la relation d'héritage entre interfaces. Un intérêt de la définition d'interfaces est leur utilisation conjointe avec le mécanisme de modules. Ainsi il est possible de construire la signature d'un module, utilisant des types objets, en donnant uniquement la description des interfaces des classes.

Déclarations locales dans les classes

Une déclaration de classe produit un type et un constructeur. Nous avons jusqu'à maintenant, pour la clarté de l'exposé, présenté ces constructeurs comme des fonctions sans environnement. En fait il est possible d'une part de définir des constructeurs qui n'ont pas besoin de valeurs initiales pour créer une instance. C'est-à-dire qui ne sont plus fonctionnels. D'autre part, il est possible d'effectuer des déclarations locales dans la classe. Ces valeurs peuvent être capturées dans le constructeur, permettant à ces valeurs d'être partagées par toutes les instances de la classe. Ces valeurs peuvent être considérées comme des variables de classe.

Constructeurs constants

Une déclaration de classe n'utilise pas forcément des valeurs initiales qui seront passées au constructeur. Par exemple la classe suivante :

```
# class exemple1 =
  object
  method print () = ()
  end ;;
class exemple1 : object method print : unit -> unit end
# let p = new exemple1 ;;
val p : exemple1 = <obj>
```

Le constructeur d'instances est constant. L'allocation n'a pas besoin de valeurs initiales pour les variables d'instance. En règle générale, il est préférable d'utiliser une valeur initiale comme () pour conserver le caractère fonctionnel du constructeur.

Déclarations locales pour les constructeurs

Une déclaration de classe peut s'écrire en utilisant directement l'abstraction.

```
# class example2 =
  fun a ->
    object
    val mutable r = a
    method get_r = r
    method plus x = r <- r + x
    end ;;
class example2 :
  int ->
  object val mutable r : int method get_r : int method pl
```

On s'aperçoit mieux du caractère fonctionnel du constructeur. Le constructeur est donc une fermeture. Celle-ci peut posséder un environnement qui lie des variables libres à un environnement de déclarations. La syntaxe des déclarations de classes autorise les déclarations locales à cette expression fonctionnelle.

Variables de classes

On appelle variables de classe, les déclarations connues au niveau

de la classe, donc partagées par toutes les instances de cette classe. Le plus souvent ces variables de classe peuvent être utilisées en dehors de toute création d'instances. En Objective CAML, on peut faire partager des valeurs, en particulier modifiables, par toutes les instances d'une classe grâce au caractère fonctionnel du constructeur qui possède alors un environnement non vide.

Nous illustrons cette possibilité par l'exemple suivant qui permet de tenir à jour le nombre d'instances d'une classe. Pour cela on définit une classe abstraite paramétrée 'a om.

```
# class virtual ['a] om =  
  object  
  method finalize () = ()  
  method virtual destroy : unit -> unit  
  method virtual to_string : unit -> string  
  method virtual all : 'a list  
end;;
```

Puis on déclare la classe 'a lo dont la fonction de construction contient les déclarations locales de n pour associer un numéro unique à chaque instance et de l qui contient la liste des couples (numéro, instance) de chaque instance encore active.

```
# class ['a] lo =  
  let l = ref []  
  and n = ref 0 in  
  fun s ->  
    object (self:'b )  
    inherit ['a] om  
    val mutable num = 0  
    val nom = s  
    method to_string () = s  
    method print () = print_string s  
    method print_all () =  
      List.iter (function (a,b) ->  
        Printf.printf "(%d,%s) " a (b#to_string())) !l  
    method destroy () = self#finalize();  
    l:= List.filter (function (a,b) -> a <> num) !l; ()  
    method all = List.map snd !l  
    initializer incr n; num <- !n; l:= (num, (self :> 'a
```

```

end;;
class ['a] lo :
  string ->
  object
  constraint 'a = 'a om
  val nom : string
  val mutable num : int
  method all : 'a list
  method destroy : unit -> unit
  method finalize : unit -> unit
  method print : unit -> unit
  method print_all : unit -> unit
  method to_string : unit -> string
end

```

À chaque création d'une instance de la classe *lo*, l'initialisateur incrémente la référence *n* et ajoute le couple (numéro, self) à la liste *l*. Les méthodes *print* et *print_all* affichent respectivement l'instance réceptrice et toutes les instances de la classe contenues dans *l*.

```

# let m1 = new lo "début";;
val m1 : ('a om as 'a) lo = <obj>
# let m2 = new lo "entre";;
val m2 : ('a om as 'a) lo = <obj>
# let m3 = new lo "fin";;
val m3 : ('a om as 'a) lo = <obj>
# m2#print_all();;
(3,fin) (2,entre) (1,début) - : unit = ()
# m2#all;;
- : ('a om as 'a) list = [<obj>; <obj>; <obj>]

```

La méthode *destroy* enlève une instance de la liste des instances créées et appelle la méthode *finalize* pour effectuer une dernière action sur cette instance avant sa disparition de la liste. La méthode *all* retourne toutes les instances d'une classe créées en utilisant **new**.

```

# m2#destroy();;
- : unit = ()
# m1#print_all();;
(3,fin) (1,début) - : unit = ()
# m3#all;;

```



```
- : ('a om as 'a) list = [<obj>; <obj>]
```

Il est à noter que les instances des sous-classes sont aussi conservées dans cette liste. Rien n'empêche d'utiliser la même technique en particulierisant certaines de ces sous-classes. Par contre les instances obtenues par copie (Oo.copy ou {< >}) ne sont pas concernées.

Exercices

Piles en objet

On reprend l'exemple des piles vu précédemment à la sauce objet.

1. Définir une classe `intpile` en utilisant les listes d'Objective CAML implantant des listes d'entiers et disposant des méthodes `empile`, `depile`, `sommet` et `taille`.

```
# exception PileVide

class intpile () =
  object
    val p = ref ([] : int list)
    method empile i = p := i:: !p
    method depile () = if !p = [] then raise PileVide
    method sommet () = if !p = [] then raise PileVide
    method taille () = List.length !p
  end ;;
exception PileVide
class intpile :
  unit ->
  object
    val p : int list ref
    method depile : unit -> unit
    method empile : int -> unit
    method sommet : unit -> int
    method taille : unit -> int
```

end

2. Créer une **instance** contenant 3 et 4 comme éléments de la pile.

```
# let p = new intpile () ;;
val p : intpile = <obj>
# p#empile 3 ;;
- : unit = ()
# p#empile 4 ;;
- : unit = ()
```

3. Définir une nouvelle classe **pile** contenant des éléments répondant à la méthode `print : unit -> unit`.

```
# class pile () =
  object
    val p = ref ([] : <print : unit -> unit> list)
    method empile i = p := i:: !p
    method depile () = if !p = [] then raise PileVide
    method sommet () = if !p = [] then raise PileVide
    method taille () = List.length !p
  end ;;
class pile :
  unit ->
  object
    val p : < print : unit -> unit > list ref
    method depile : unit -> unit
    method empile : < print : unit -> unit > -> unit
    method sommet : unit -> < print : unit -> unit >
    method taille : unit -> int
  end
```

4. Définir une classe paramétrée [*a*] **pile** avec les mêmes méthodes.

```

# class ['a] ppile () =
  object
  val p = ref ([] : 'a list)
  method empile i = p := i:: !p
  method depile () = if !p = [] then raise PileVide
  method sommet () = if !p = [] then raise PileVide
  method taille () = List.length !p
  end ;;
class ['a] ppile :
  unit ->
  object
  val p : 'a list ref
  method depile : unit -> unit
  method empile : 'a -> unit
  method sommet : unit -> 'a
  method taille : unit -> int
  end

```

5. Comparer les différentes classes de piles.

Liaison retardée

Cet exercice illustre la notion de liaison retardée indépendamment de la notion de sous-typage.

Soit le programme donné par la suite :

1. Dessiner les relations entre classes
2. Tracer les différents envois de messages.
3. En supposant être dans un mode caractère sans écho, indiquer ce qu'affiche le programme.

```

exception CrLf;;
class lecture_chaine (m) =
  object (self)

```

```

val msg = m
val mutable res = ""

method lire_char =
let c = input_char stdin in
if (c != '\n') then begin
output_char stdout c; flush stdout
end;
String.make 1 c

method private lire_chaine_aux =
while true do
let s = self#lire_char in
if s = "\n" then raise CrLf
else res <- res ^ s;
done

method private lire_chaine_aux2 =
let s = self#lire_char in
if s = "\n" then raise CrLf
else begin res <- res ^ s; self#lire_chaine_aux2 end

method lire_chaine =
try
self#lire_chaine_aux
with End_of_file -> ()
| CrLf -> ()

method input = res <- ""; print_string msg; flush stdout
self#lire_chaine

method get = res
end;;

class lecture_mdp (m) =
object (self)
inherit lecture_chaine m
method lire_char = let c = input_char stdin in
if (c != '\n') then begin
output_char stdout '*'; flush stdout

```

```

end;

let s = " " in s.[0] <- c; s
end;;

let login = new lecture_chaine("Login : ");;
let passwd = new lecture_mdp("Passwd : ");;
login#input;;
passwd#input;;
print_string (login#get);;print_newline();;
print_string (passwd#get);;print_newline();;

```

Classes abstraites et évaluateur d'expressions

Cet exercice a pour but de montrer la factorisation de code réalisée par l'utilisation de classes abstraites.

Les expressions arithmétiques construites sont toutes des instances d'une sous-classe de la classe abstraite *expr_ar*.

1. Définir une classe abstraite *expr_ar* pour les expressions arithmétiques contenant deux méthodes abstraites : *eval* de type *float* et *print* de type *unit* qui respectivement évalue et affiche une expression arithmétique.

```

# class virtual expr_ar () =
  object
    method virtual eval : unit -> float
    method virtual print : unit -> unit
  end ;;
class virtual expr_ar :
  unit ->
  object
    method virtual eval : unit -> float
    method virtual print : unit -> unit
  end

```

2. Définir une classe concrète *constante* sous-classe de *expr_ar*.

```
# class constante x =  
  object  
  inherit expr_ar ()  
  val c = x  
  method eval () = c  
  method print () = print_float c  
  end ;;
```

```
class constante :  
  float ->  
  object  
  val c : float  
  method eval : unit -> float  
  method print : unit -> unit  
  end
```

(* autre solution : *)

```
# class const x =  
  object  
  inherit expr_ar ()  
  method eval () = x  
  method print () = print_float x  
  end ;;
```

```
class const :  
  float -> object method eval : unit -> float method
```

3. Définir une sous-classe abstraite *bin_op* de *expr_ar* implantant les méthodes *eval* et *print* en utilisant deux nouvelles méthodes abstraites *oper* de type $(float * float) \rightarrow float$ (utilisée par *eval*) et *symbole* de type *string* (utilisée par *print*).

```
# class virtual bin_op g d =  
  object (this)  
  inherit expr_ar ()  
  val fg = g
```

```

val fd = d
method virtual symbole : string
method virtual oper : float * float -> float
method eval () =
let x = fg#eval()
and y = fd#eval() in
  this#oper(x,y)
method print () =
  fg#print () ;
  print_string (this#symbole) ;
  fd#print ()
end ;;
class virtual bin_op :
  (< eval : unit -> float; print : unit -> 'b; .. >
  (< eval : unit -> float; print : unit -> unit; ..
  object
  val fd : 'c
  val fg : 'a
  method eval : unit -> float
  method virtual oper : float * float -> float
  method print : unit -> unit
  method virtual symbole : string
  end

```

4. Définir les classes concrètes *add* et *mul* sous-classes de *bin_op* implantant chacune les méthodes *oper* et *symbole*.

```

# class add x y =
  object
    inherit bin_op x y
    method symbole = "+"
    method oper(x,y) = x +. y
    end ;;
class add :
  (< eval : unit -> float; print : unit -> 'b; .. >
  (< eval : unit -> float; print : unit -> unit; ..
  object
  val fd : 'c

```

```

val fg : 'a
method eval : unit -> float
method oper : float * float -> float
method print : unit -> unit
method symbole : string
end

# class mul x y =
object
inherit bin_op x y
method symbole = "*"
method oper(x,y) = x *. y
end ;;
class mul :
  (< eval : unit -> float; print : unit -> 'b; .. >
  (< eval : unit -> float; print : unit -> unit; ..
  object
  val fd : 'c
  val fg : 'a
  method eval : unit -> float
  method oper : float * float -> float
  method print : unit -> unit
  method symbole : string
  end

```

5. Dessiner l'arbre d'héritage.

6. Écrire une [fonction](#) qui prenant une suite de Genlex.token construit un objet de type *expr_ar*.

```

# open Genlex ;;
# exception Found of expr_ar ;;
exception Found of expr_ar

# let rec create accu l =
  let r = match Stream.next l with
  Float f -> new constante f
  | Int i -> ( new constante (float i) :> expr_ar)

```



```

| Kwd k  ->
let v1 = accu#somet() in accu#depile();
let v2 = accu#somet() in accu#depile();
( match k with
  "+" -> ( new add v2 v1 :> expr_ar)
|  "*" -> ( new mul v2 v1 :> expr_ar)
|  ";" -> raise (Found (accu#somet()))
|  _ -> failwith "aux : bas keyword"
|  _ -> failwith "aux : bad case"
in
  create (accu#empile (r :> expr_ar); accu) 1 ;;
val create :
  < depile : unit -> 'a; empile : expr_ar -> 'b; som
  .. > ->
  Genlex.token Stream.t -> 'c = <fun>

# let gl = Genlex.make_lexer ["+"; "*"; ";"] ;;
val gl : char Stream.t -> Genlex.token Stream.t = <

# let run () =
  let s = Stream.of_channel stdin in
  create (new ppile ()) (gl s) ;;
val run : unit -> 'a = <fun>

```

7. Tester ce programme en lisant l'entrée standard, en utilisant l'analyseur lexical générique Genlex. On pourra entrer les expressions à évaluer sous forme post-fixée.

Jeu de la vie en objet

On définit les 2 classes suivantes :

- `cell` : pour les cellules du monde, répondant à la méthode `isAlive` : `unit -> bool`
- `world` : ayant un tableau de `cell` et répondant aux messages :

```

display : unit -> unit
nextGen : unit -> unit
setCell : int * int -> cell -> unit
getCell : int * int -> cell

```

1. Écrire la classe `cell`.

```

# class cell a =
  object
    val mutable v = (a : bool)
    method isAlive = v
  end ;;
class cell : bool -> object val mutable v : bool me

```

2. Écrire une classe abstraite `absWorld` qui implante les méthodes `display`, `getCell` et `setCell` et laisse abstraite la méthode `nextGen`.

```

# class virtual absWorld n m =
  object(self)
    val mutable tcell = Array.create_matrix n m (new c
    val maxx = n
    val maxy = m
    val mutable gen = 0
    method private dessine(c) =
      if c#isAlive then print_string "*"
      else print_string "."
    method display() =
      for i = 0 to (maxx-1) do
        for j=0 to (maxy -1) do
          print_string " " ;
          self#dessine(tcell.(i).(j))
        done ;
      print_newline()
    done
    method getCell(i,j) = tcell.(i).(j)
    method setCell(i,j,c) = tcell.(i).(j) <- c
    method getCells = tcell

```

```

end ;;
class virtual absWorld :
  int ->
  int ->
  object
  val mutable gen : int
  val maxx : int
  val maxy : int
  val mutable tcell : cell array array
  method private dessine : cell -> unit
  method display : unit -> unit
  method getCell : int * int -> cell
  method getCells : cell array array
  method setCell : int * int * cell -> unit
end

```

3. Écrire une classe **world**, sous-classe de `absWorld`, qui implante la méthode `nextGen` selon les règles de croissance.

```

# class world n m =
  object (self)
    inherit absWorld n m
    method neighbors(x,y) =
      let r = ref 0 in
      for i=x-1 to x+1 do
        let k = (i+maxx) mod maxx in
        for j=y-1 to y+1 do
          let l = (j + maxy) mod maxy in
          if tcell.(k).(l)#isAlive then incr r
        done
      done;
    if tcell.(x).(y)#isAlive then decr r ;
    !r

  method nextGen() =
    let w2 = new world maxx maxy in
    for i=0 to maxx-1 do
      for j=0 to maxy -1 do

```

```

let n = self#neighbors(i,j) in
if tcell.(i).(j)#isAlive
then (if (n = 2) || (n = 3) then w2#setCell(i,j,new cell true))
else (if n = 3 then w2#setCell(i,j,new cell true))
done
done ;
tcell <- w2#getCells ;
gen <- gen + 1
end ;;
class world :
  int ->
  int ->
  object
  val mutable gen : int
  val maxx : int
  val maxy : int
  val mutable tcell : cell array array
  method private dessine : cell -> unit
  method display : unit -> unit
  method getCell : int * int -> cell
  method getCells : cell array array
  method neighbors : int * int -> int
  method nextGen : unit -> unit
  method setCell : int * int * cell -> unit
end

```

4. Écrire le **programme principal** qui crée un monde vide, lui ajoute certaines cellules, puis rentre dans une boucle d'interaction qui affiche le monde, attend une interaction, calcule ensuite la génération suivante et repart dans la boucle.

```

# exception Fin;;
exception Fin

# let main () =
  let a = 10 and b = 12 in
  let w = new world a b in
  w#setCell(4,4,new cell true) ;

```

```

w#setCell(4,5,new cell true) ;
w#setCell(4,6,new cell true) ;
try
while true do
w#display() ;
if ((read_line()) = "F") then raise Fin else w#next
done
with Fin -> () ;;
val main : unit -> unit = <fun>

```

Résumé

Ce chapitre a décrit l'extension objet du langage Objective CAML. La structuration en classes est une alternative aux modules qui permet d'une part la modélisation objet d'une application et d'autre part la réutilisation et la modifiabilité des programmes grâce à l'héritage et la liaison retardée. Cette extension s'intègre au système de types d'Objective CAML et lui ajoute la notion de sous-type, permettant ainsi d'utiliser une instance d'un sous-type en tout endroit où une valeur du type est attendue. En combinant le sous-typage et la liaison retardée, on obtient un polymorphisme d'inclusion permettant, par exemple, de construire des listes homogènes du point de vue des types, mais hétérogènes du point de vue des comportements.

Pour en savoir plus

La littérature sur la programmation par objets et les langages par objets est considérable, chaque langage objet implantant un modèle différent.

Un ouvrage général et toujours d'actualité pour sa première partie, est « Langages à Objets » ([[MNC+89](#)]) qui explique la démarche objet. Un livre plus pointu, « Langages et modèles à objets » [[DEMN98](#)], fait un état de l'art du domaine.

Sur la modélisation, l'ouvrage < > ([[GHJV98](#)]) propose un catalogue de modèles de conception donnant une bonne description

des possibilités de réutilisabilité.

Pour la notation UML, le site de référence est celui de Rational :

Lien: <http://www.rational.com/uml/resources>

Pour les langages fonctionnels avec une extension objet, on peut citer les `< >` objets, provenant du monde SMALLTALK, et CLOS (pour Common Lisp Object System) ainsi que de nombreux Scheme implantant les fonctions génériques à la CLOS.

D'autres propositions de langages par objets ont été faites pour les langages fonctionnels typés dynamiquement, comme par exemple Haskell, langage fonctionnel pur, qui autorise le polymorphisme paramétrique et *ad hoc* pour la surcharge.

L'article [RV98] présente l'extension objet d'Objective CAML d'un point de vue théorique.

Plusieurs cours en ligne peuvent être consultés pour approfondir le type statique objet d'Objective CAML.

Le cours de María-Virginia Aponte.

Lien: <http://tulipe.cnam.fr/personne/aponte/ocaml.html>

Une courte présentation des objets par Didier Rémy.

Lien: <http://pauillac.inria.fr/~remy/objectdemo.html>

Le cours de Didier Rémy au Magistère MMFAI

Lien: <http://pauillac.inria.fr/~remy/classes/magistere/>

Le cours de Roberto Di Cosmo au Magistère MMFAI.

Lien: <http://www.dmi.ens.fr/users/dicosmo/CourseNotes/OO/>

Comparaison des modèles d'organisation

Introduction

Les chapitres 14 et 15 ont respectivement présenté deux modèles d'organisation d'applications : le modèle fonctionnel/modulaire et le modèle objet. Ces deux modèles répondent, chacun à leur manière, aux besoins du développement d'une application :

- organisation logique d'un programme : module ou classe
- compilation séparée : module simple
- types abstraits de données : module (type abstrait) ou objet
- réutilisation des composants : foncteurs/partage de types avec polymorphisme paramétrique ou héritage/sous-typage avec des classes paramétrées
- modifiabilité des composants : liaison tardive (objet)

Le développement d'une application modulaire commence par son découpage en unités logiques : les modules. Ensuite vient la réalisation de leur spécification en écrivant leur signature, et en dernier l'implantation. Lors de la réalisation d'un module, il peut être nécessaire de modifier sa signature ou celle de ses paramètres; il faut alors modifier les sources de celui-ci. Ceci est peu satisfaisant si ce même module est déjà utilisé par une autre application. Néanmoins cette démarche offre un cadre strict et rassurant pour le programmeur.

Dans le modèle objet, l'analyse d'un problème aboutit à la description de relations entre classes. Si par la suite une classe n'offre pas les services attendus, il est toujours possible de la sous-classer pour l'étendre. Cette démarche permet de récupérer de grandes hiérarchies de classes sans modifier leurs sources, ne modifiant pas ainsi le comportement d'une autre application l'utilisant. Malheureusement cette technique est inflationniste et pose des difficultés de duplication avec l'héritage multiple.

De nombreux problèmes nécessitent des types de données récursifs et des opérations qui manipulent des valeurs de ces types. Il arrive souvent que le problème évolue, soit en cours de réalisation, soit en maintenance, impliquant une extension des types et des opérations. Aucun de ces deux modèles ne permet l'extension dans les deux sens. En fonctionnel/modulaire, les types ne sont pas extensibles,

par contre on peut créer de nouvelles fonctions (opérations) sur ces types. En objet, on peut étendre les données mais pas les traitements (en créant une nouvelle sous-classe d'une classe abstraite qui implante ses méthodes). En cela ces deux modèles sont duaux.

L'avantage de réunir ces deux modèles dans le même langage est de pouvoir choisir le modèle le plus approprié pour la résolution du problème posé et de les mixer dans le but de dépasser les limitations de chaque modèle.

Plan du chapitre

La première section compare le modèle modulo-fonctionnel et le modèle objet. Cette comparaison distingue les points particuliers de chaque modèle, pour montrer ensuite comment plusieurs d'entre eux peuvent se traduire à la main dans l'autre modèle. On peut ainsi simuler la relation d'héritage dans les modules et utiliser les classes pour traduire des modules simples. Les limitations de chaque modèle sont ensuite récapitulées. La deuxième section s'intéresse au problème d'extensibilité pour les structures de données et les traitements, et propose une solution qui mixe les deux modèles. La troisième section décrit d'autres compositions des deux modèles par l'utilisation des types abstraits des modules pour les objets.

Comparaison modules et objets

La principale différence entre programmation modulaire et programmation objet en Objective CAML provient du système de types. En effet la programmation par modules reste dans le système de types à la ML, i.e. du polymorphisme paramétrique (même code exécuté pour différents types de paramètres), alors que la programmation par objets entraîne un polymorphisme ad hoc (où l'envoi d'un message à un objet déclenche l'application de codes différents). Cela est particulièrement clair avec le sous-typage. Cette extension du système de types à la ML ne peut être simulée en ML pur. Il sera toujours impossible de construire des listes hétérogènes sans casser le système de types.

La programmation modulaire et la programmation par objets sont deux réponses sûres (grâce au typage) à l'organisation logique d'un programme, permettent la réutilisabilité et la modifiabilité de composants logiciels. La programmation par objets en Objective CAML autorise le polymorphisme paramétrique (classes paramétrées) et d'*inclusion* (envoi de messages) grâce à la liaison tardive et au sous-typage, avec des restrictions dues à l'égalité, ce qui facilite une programmation incrémentielle. La programmation modulaire permet de restreindre le polymorphisme paramétrique et utilise la liaison immédiate ce qui peut être utile pour conserver l'efficacité de l'exécution.

Le modèle de la programmation modulaire permet de facilement étendre les fonctionnalités sur des types de données récurifs non extensibles. Si l'on désire ajouter un cas dans un type somme, il sera nécessaire de modifier une bonne partie des sources.

Le modèle de la programmation objet définit un ensemble récursif de données à partir de classes. On prend une classe par cas de données.

Efficacité d'exécution

La liaison retardée correspond à une indirection dans le tableau des méthodes (voir page ??). Comme l'accès, de l'extérieur de la classe, à une variable d'instance passe aussi par un envoi de message, cette accumulation d'indirections peut se révéler coûteuse.

Pour montrer cette perte d'efficacité, nous construisons la hiérarchie de classe suivante :

```
# class virtual test () =  
  object  
    method virtual somme : unit -> int  
    method virtual somme2 : unit -> int  
  end;;  
# class a x =  
  object (self)  
    inherit test ()  
  val a = x
```

```

method a = a
method somme () = a
method somme2 () = self#a
end;;
# class b x y =
  object(self)
    inherit a x as super
    val b = y
    method b = b
    method somme () = b + a
    method somme2 () = self#b + super#somme2()
    end;;

```

On compare ensuite les temps d'exécution entre d'une part l'envoi des messages `somme`/`somme2` sur une instance de la classe `b` et d'autre part l'appel à la fonction `f` suivante.

```

# let f a b = a + b ;;
# let iter g a n = for i = 1 to n do ignore(g a) done ;
# let go i j = match i with
  1 -> iter (fun x -> x#somme()) (new b 1 2) j
  | 2 -> iter (fun x -> x#somme2()) (new b 1 2) j
  | 3 -> iter (fun x -> f 1 x) 2 j ;;

# go (int_of_string (Sys.argv.(1))) (int_of_string (Sys.

```

Pour 10 millions d'itérations, on obtient les résultats suivants :

	code-octet	natif
cas 1	07,5 s	0,6 s
cas 2	15,0 s	2,3 s
cas 3	06,0 s	0,3 s

Cet exemple a été construit dans le but de montrer que la liaison retardée a un coût par rapport à la liaison statique habituelle. Ce coût dépend de la quantité de calcul par rapport au nombre d'envois de messages dans une méthode. L'utilisation du compilateur natif diminue la partie calcul sans changer la partie indirection du test. On s'aperçoit dans le cas 2 que les multiples indirections à l'envoi du message `somme2` ont un coût incompressible.

Exemple : interface graphique

La bibliothèque graphique UPI (voir page ??) a été conçue en utilisant le noyau fonctionnel/impératif du langage. Elle s'adapte très facilement sous forme de modules. Chaque composant devient un module indépendant, permettant ainsi une harmonisation des noms des fonctions. Pour ajouter un composant, il est nécessaire de connaître le type concret des composants. Au nouveau module de modifier les champs nécessaires pour décrire son affichage et ses comportements.

Elle peut aussi se réécrire en objet. Pour cela on construit la hiérarchie de classes de la figure 16.1.

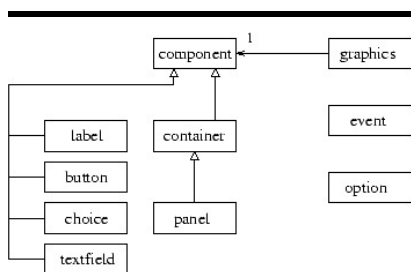


Figure 16.1 : hiérarchie de classes pour Upi

Il est plus aisé d'ajouter de nouveaux composants grâce à l'héritage qu'en utilisant les modules. Néanmoins, l'absence de surcharge nécessite toujours un codage des options en tant que paramètres des méthodes. L'utilisation de la relation de sous-typage permet de facilement construire la liste des composants d'un conteneur. La liaison retardée sélectionne les méthodes propres au composant. L'intérêt de l'objet provient aussi de la possibilité d'étendre ou de modifier le contexte graphique, et les autres types utilisés, toujours grâce à l'héritage. C'est pour cela que les principales bibliothèques graphiques sont organisées selon le modèle objet.

Traduction de modules en classes

Un module simple ne déclarant qu'un seul type et ne possédant pas de fonction polymorphe indépendante du type déclaré peut se traduire en une classe. Selon la nature du type utilisé (type enregistrement ou type somme) on traduit le module en classe de manière différente.

Déclarations de types

Type enregistrement

Un type enregistrement s'écrit directement sous forme d'une classe où chaque champ de l'enregistrement devient une variable d'instance.

Type somme

Un type somme se traduit en plusieurs classes en utilisant le modèle de conception `< >`. Une classe abstraite décrit les opérations (les fonctions) sur ce type. Chaque branche du type somme devient alors une sous-classe de la classe abstraite et implante les méthodes abstraites pour sa branche. Il n'y a plus alors de `< >` mais choix de la méthode spécifique à la branche.

Types paramétrés

Les types paramétrés s'implantent par des classes paramétrées.

Types abstraits

On peut considérer une classe comme un type abstrait. À aucun moment l'état interne de la classe n'est visible à l'extérieur de sa hiérarchie. Néanmoins rien n'empêche de définir une sous-classe

pour accéder aux variables d'instances d'une classe.

Types mutuellement récursifs

Les déclarations de types mutuellement récursifs se traduisent par des déclarations de classes mutuellement récursives.

Déclarations de fonctions

Les fonctions dont un paramètre dépend du type t du module se traduisent en méthodes. Les fonctions où n'apparaît pas le type t peuvent être déclarées **private** dans la mesure où leur appartenance au module n'est pas directement liée au type t . Cela a de plus l'avantage de ne pas poser problème si des variables de types apparaissent dans le type des paramètres. Se pose alors le problème des fonctions dont un paramètre est du type t et un autre est du type ' a '. Ces fonctions sont très rares dans les modules de la bibliothèque standard. On distingue les modules $< >$ comme Marshal ou Printf qui ont un typage non standard, des modules sur les structures linéaires comme List. Pour ce dernier la fonction fold_left, de type $(a \rightarrow b \rightarrow 'a) \rightarrow 'a \rightarrow 'b \text{ list} \rightarrow 'a$, est difficilement traduisible telle quelle dans une méthode de la classe $[b] \text{ liste}$ car la variable de type ' a ' est libre et ne peut apparaître dans le type de la méthode. Plutôt que d'ajouter un paramètre de type a à la classe liste, il est préférable de sortir ces fonctions dans de nouvelles classes, paramétrées par 2 variables de types et possédant un champ liste.

Méthodes binaires

Les méthodes binaires en dehors du sous-typage ne posent pas de difficulté.

Autres déclarations

Déclarations de valeurs non fonctionnelles. On peut accepter de

déclarer les valeurs non fonctionnelles en dehors des classes. Il en est de même pour les exceptions.

Exemple : listes avec itérateur

On cherche à traduire en objet un module ayant la signature LISTE suivante :

```
# module type LISTE = sig
  type 'a liste = C0 | C1 of 'a * 'a liste
  val add : 'a liste -> 'a -> 'a liste
  val length : 'a liste -> int
  val hd : 'a liste -> 'a
  val tl : 'a liste -> 'a liste
  val append : 'a liste -> 'a liste -> 'a liste
  val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b liste -> 'b
end ;;
```

On déclare tout d'abord la classe abstraite *'a liste* correspondant à la définition du type.

```
# class virtual ['a] liste () =
  object (self : 'b)
  method virtual add : 'a -> 'a liste
  method virtual empty : unit -> bool
  method virtual hd : 'a
  method virtual tl : 'a liste
  method virtual length : unit -> int
  method virtual append : 'a liste -> 'a liste
  end ;;
```

On définit ensuite les deux sous-classes *c1_liste* et *c0_liste* pour chaque composant du type somme. Chacune de ces classes doit définir les méthodes de la classe abstraite ancêtre.

```
# class ['a] c1_liste (t, q) =
  object (self )
  inherit ['a] liste () as super
  val t = t
  val q = q
```

```

method add x = new c1_liste (x, (self : 'a #liste :> 'a
method empty () = false
method length () = 1 + q#length()
method hd = t
method tl = q
method append l  = new c1_liste (t,q#append l)
end ;;
# class ['a] c0_liste () =
object (self)
inherit ['a] liste () as super
method add x = new c1_liste (x, (self : 'a #liste :> 'a
method empty () = true
method length () = 0
method hd = failwith "c0_liste : hd"
method tl = failwith "c0_liste : tl"
method append l  = l
end ;;
# let l = new c1_liste (4, new c1_liste (7, new c0_liste
val l : int liste = <obj>

```

La fonction `LISTE.fold_left` n'a pas été incorporée dans la classe `liste` pour éviter d'introduire un nouveau paramètre de type. On préfère définir la classe `fold_left` pour implanter ce traitement. Pour cela, on utilise une variable d'instance fonctionnelle (`f`).

```

# class virtual ['a,'b] fold_left () =
object (self)
method virtual f : 'a -> 'b -> 'a
method iter r (l : 'b liste) =
if l#empty() then r else self#iter (self#f r (l#hd)) (l
end ;;
# class ['a,'b] gen_fl f =
object
inherit ['a,'b] fold_left ()
method f = f
end ;;

```

On construit alors une instance de la classe `gen_fl` pour l'addition :

```

# let afl = new gen_fl (+) ;;
val afl : (int, int) gen_fl = <obj>

```

```
# afl#iter 0 1 ;;
- : int = 11
```

Simulation d'héritage avec des modules

La relation d'héritage entre classes permet de récupérer dans une sous-classe l'ensemble des déclarations de variables et de méthodes de la classe ancêtre. On peut simuler cette relation en utilisant des modules. La sous-classe qui hérite se transforme en un module paramétré dont le paramètre est la classe ancêtre. L'héritage multiple augmente le nombre de paramètres du module. On reprend l'exemple classique sur les points et les points colorés décrit au chapitre 15 pour le traduire en modules.

La classe point devient le module Point de signature POINT suivante.

```
# module type POINT =
  sig
    type point
    val new_point : (int * int) -> point
    val get_x : point -> int
    val get_y : point -> int
    val moveto : point -> (int * int) -> unit
    val rmoveto : point -> (int * int) -> unit
    val display : point -> unit
    val distance : point -> float
  end ;;
```

La classe point_coleur se transforme en un module paramétré PointColore dont le paramètre est de signature POINT.

```
# module PointColore = functor (P : POINT) ->
  struct
    type point_coleur = {p:P.point;c:string}
    let new_point_coleur p c = {p=P.new_point p;c=c}
    let get_c self = self.c
    (* début *)
```



```

let get_x self = let super = self.p in P.get_x super
let get_y self = let super = self.p in P.get_y super
let moveto self = let super = self.p in P.moveto super
let rmoveto self = let super = self.p in P.rmoveto super
let display self = let super = self.p in P.display super
let distance self = let super = self.p in P.distance super
(* fin *)
let display self =
let super = self.p in P.display super; print_string ("o
end ;;

```

La lourdeur des déclarations `< >` peut être allégée par une procédure automatique de traduction ou une extension du langage. Les déclarations récursives de méthodes pourraient s'écrire par un seul `let rec ... and`. L'héritage multiple entraîne des foncteurs à plusieurs paramètres. Le coût de la redéfinition n'est pas plus grand que la liaison tardive.

La liaison tardive n'est pas implantée dans cette simulation. Pour le faire, il faut définir un enregistrement où chaque champ correspond au type des fonctions/méthodes.

Limitations de chaque modèle

Le modèle fonctionnel/modulaire offre un cadre rassurant mais rigide pour la modifiabilité du code. Le modèle objet d'Objective CAML souffre de la double vision des classes : structuration et type, impliquant l'absence de surcharge et l'impossibilité d'effectuer des contraintes de type d'un type ancêtre vers un type descendant.

Modules

Les principales limitations du modèle fonctionnel/modulaire proviennent de la difficulté d'extensibilité des types. Bien que les types abstraits permettent de s'affranchir de la représentation concrète d'un type, leur utilisation dans les modules paramétrés nécessite d'indiquer à la main les égalités de types entre modules, compliquant l'écriture des signatures.

Dépendances récursives

Le graphe de dépendances des modules d'une application ne contient pas de cycle (DAG). Cela implique d'une part de ne pas avoir de types mutuellement récursifs entre deux modules et d'autre part d'empêcher les déclarations de valeurs mutuellement récursives.

Difficultés d'écriture des signatures

Un des intérêts de l'inférence de types est de ne pas nécessiter l'écriture des types des paramètres des fonctions. La description des signatures fait perdre cette pratique. Il devient nécessaire de spécifier les types des déclarations de la signature «à la main». On peut utiliser l'option `-i` du compilateur `ocamlc` pour afficher le type de toutes les déclarations globales d'un fichier `.ml` et utiliser cette information pour construire la signature d'un module. Dans ce cas, on perd la démarche `< >` qui consiste à spécifier le module avant de l'implanter. De plus si la signature et le module subissent des changements importants, on doit revenir à l'édition de la signature. Les modules paramétrés ont besoin des signatures de leurs paramètres et celles-ci doivent aussi être décrites manuellement. Enfin si on associe une signature fonctionnelle à un module paramétré, il n'existe pas de possibilité pour récupérer la signature résultat de l'application du foncteur. Cela oblige à écrire principalement des signatures non fonctionnelles, quitte à les assembler par la suite pour construire une signature fonctionnelle.

Exportation et importation de modules

L'importation des déclarations d'un module simple s'effectue soit par la notation pointée (`Module.nom`), soit directement par le nom d'une déclaration (`nom`) si le module a été ouvert (`openModule`). Les déclarations de l'interface du module importé ne sont pas directement exportables au niveau du module en cours de définition. Celui-ci a accès à ces déclarations, mais elles ne sont pas considérées comme des déclarations du module. Pour ce faire il est

nécessaire de déclarer, à la manière de la simulation d'héritage, les valeurs importées. Il en est de même pour les modules paramétrés. Les déclarations du module paramètre ne sont pas considérées comme des déclarations du module courant.

Objets

Les principales limitations du modèle objet d'Objective CAML proviennent du typage.

- pas de méthode contenant un paramètre de type libre;
- difficulté d'échappement du type de la classe dans une de ses méthodes;
- absence de contrainte de type d'ancêtre vers descendant;
- pas de surcharge.

Le point le plus déroutant quand on aborde l'extension objet d'Objective CAML provient de l'impossibilité de construire des méthodes contenant un type paramétré dont le paramètre de type est libre. La déclaration d'une classe peut être vue comme la définition d'un nouveau type et suit en cela la règle générale qui interdit la présence de variables de type libres dans une déclaration de type. C'est pour cela que les classes paramétrées sont indispensables dans le modèle objet d'Objective CAML car elles permettent de lier ces variables de type.

Absence de surcharge

Le modèle objet d'Objective CAML ne permet pas la surcharge de méthode. Comme le type d'un objet correspond aux types de ces méthodes, le fait de posséder plusieurs méthodes de même nom mais de types différents entraîne de nombreuses ambiguïtés, dues au polymorphisme paramétrique, que le système ne pourrait résoudre que dynamiquement, ce qui est contradictoire avec la vision totalement statique du typage. Prenons une classe exemple qui possède deux méthodes message possédant pour la première un paramètre entier et pour la deuxième un paramètre flottant. Soit e une instance de cette classe et soit la fonction f suivante :

```
# let f x a = x#message a ;;
```

Les appels `f e 1` et `f e 1.1` ne peuvent pas statiquement être résolus car il n'y a pas dans le code de la fonction `f` d'informations spécifiques à la classe `exemple`.

Une conséquence immédiate de cette absence est l'unicité des constructeurs d'instances. La déclaration d'une classe indique les paramètres à fournir à la fonction de création. Celle-ci est unique.

Initialisation

L'initialisation des variables d'instance déclarées dans une classe peut être problématique quand elle doit se calculer à partir des valeurs passées à la construction.

Égalité entre instances

La seule égalité qui fonctionne sur les objets est l'égalité physique. L'égalité structurelle retourne toujours `false` quand elle s'applique à deux objets physiquement différents. Cela peut surprendre dans la mesure où deux instances d'une même classe partagent la même table des méthodes. On pourrait imaginer un test physique sur la table des méthodes et un test structurel sur les valeurs (`val`) des objets. Ce sont des choix d'implantation à la manière du filtrage linéaire.

Hérarchie de classes

On ne trouve aucune hiérarchie de classes dans la distribution du langage. En effet l'ensemble des bibliothèques sont fournies sous forme de modules simples ou paramétrés. Cela montre que l'extension objet du langage est encore en cours de stabilisation et milite peu en faveur de son emploi intensif.

Étendre les composants

On appelle composant un regroupement de données et de traitements sur ces données. Dans le modèle fonctionnel/modulaire, un composant comprend la définition d'un type et des fonctions manipulant ce type. De même un composant dans le modèle objet correspond à une hiérarchie de classes, héritant d'une même classe et donc possédant tous les comportements de celle-ci. Le problème de l'extensibilité des composants consiste à vouloir d'une part augmenter les traitements/comportements sur ceux-ci et d'autre part d'augmenter les données traitées et cela sans modifier les programmes sources initiaux. Par exemple un composant image peut être soit un rectangle, soit un cercle que l'on peut afficher ou déplacer.

	rectangle	cercle	groupe
affiche	X	X	
déplace	X	X	
agrandit			

On peut vouloir étendre le composant image par le traitement agrandir et créer des groupes d'images. Le comportement des deux modèles diffère selon le sens de l'extension : données ou traitements. On définit tout d'abord, dans chaque modèle, la partie commune du composant image, puis on cherche à l'étendre.

En fonctionnel

On définit le type *image* comme un type somme contenant deux cas. Les traitements prennent en entrée un paramètre de type *image* et effectuent le travail adéquat.

```
# type image = Rect of float | Cercle of float ;;  
# let affiche = function Rect r -> ... | Cercle c -> ...  
# let déplace = ... ;;
```

On encapsule ensuite ces déclarations globales dans un module simple.

Extension des traitements

L'extension des traitements dépend de la représentation du type *image* dans le module. Si celui-ci est abstrait, il n'y a plus de possibilité d'extension des traitements. Dans le cas où le type est resté concret, il est facile d'ajouter une fonction agrandir qui effectue le changement d'échelle d'une image en sélectionnant un rectangle ou un cercle par filtrage de motifs.

Extension des données

L'extension des données ne peut pas s'effectuer avec le type *image*. En effet les types Objective CAML ne sont pas extensibles, sauf pour le type *exn* représentant des exceptions. Il n'est pas possible d'étendre les données en conservant le même type, il est donc nécessaire de définir un nouveau type *n_image* de la manière suivante :

```
type n_image = I of image | G of n_image * n_image;;
```

On doit redéfinir alors les traitements pour ce nouveau type en simulant une sorte d'héritage. Cela devient complexe dès qu'il y a plusieurs extensions.

En objet

On définit les classes rectangle et cercle, sous-classes de la classe abstraite image qui a deux méthodes abstraites affiche et deplace.

```
# class virtual image () =  
  object (self:'a)  
    method virtual affiche : unit -> unit  
    method virtual deplace : float * float -> unit  
  end;;  
# class rectangle x y w h =  
  object  
    inherit image ()  
    val mutable x = x
```

```

val mutable y = y
val mutable w = w
val mutable h = h
method affiche () = Printf.printf "R: (%f,%f) [%f,%f]"
method deplace (dx,dy) = x <- x +. dx; y <- y +. dy
end;;
# class cercle x y r =
  object
    val mutable x = x
    val mutable y = y
    val mutable r = r
    method affiche () = Printf.printf "C: (%f,%f) [%f]" x y
    method deplace (dx, dy) = x <- x +. dx; y <- y +. dy
  end;;

```

Le programme suivant construit une liste d'images et l'affiche.

```

# let r = new rectangle 1. 1. 3. 4.;;
val r : rectangle = <obj>
# let c = new cercle 1. 1. 4.;;
val c : cercle = <obj>
# let l = [ (r :> image); (c :> image)];;
val l : image list = [<obj>; <obj>]
# List.iter (fun x -> x#affiche(); print_newline()) l;;
R: (1.000000,1.000000) [3.000000,4.000000]
C: (1.000000,1.000000) [4.000000]
- : unit = ()

```

Extension des données

Les données sont facilement extensibles en ajoutant une sous-classe à la classe image de la manière suivante.

```

# class groupe i1 i2 =
  object
    val i1 = (i1:#image)
    val i2 = (i2:#image)
    method affiche () = i1#affiche(); print_newline (); i2#
    method deplace p = i1#deplace p; i2#deplace p
  end;;

```

On s'aperçoit alors que le `<> image` devient récursif car la classe *groupe* dépend hors héritage de la classe *image*.

```
# let g = new groupe (r:>image) (c:>image);;
val g : groupe = <obj>
# g#affiche();;
R: (1.000000,1.000000) [3.000000,4.000000]
C: (1.000000,1.000000) [4.000000]- : unit = ()
```

Extension des traitements

On définit une sous-classe abstraite d'*image* contenant une nouvelle méthode.

```
# class virtual e_image () =
  object
    inherit image ()
    method virtual surface : unit -> float
  end;;
```

On peut définir les classes *e_rectangle* et *e_cercle* qui héritent de *e_image* et de *rectangle* et *cercle* respectivement. On peut alors travailler sur ces images étendues pour utiliser cette nouvelle méthode. Il subsiste une difficulté pour la classe *groupe*. Celle-ci possède deux champs de type *image*, donc même en héritant de la classe *e_image* il ne sera pas possible d'envoyer le message agrandi aux champs images. Il est donc possible d'étendre les traitements sauf pour les cas de sous-classes correspondant à des types récursifs.

Extension des données et des traitements

Pour réaliser l'extension dans les deux sens, il est nécessaire de définir les types récursifs sous forme de classe paramétrée. On redéfinit la classe *groupe*.

```
# class ['a] groupe i1 i2 =
  object
    val i1 = (i1:'a)
    val i2 = (i2:'a)
```



```

method affiche () = i1#affiche(); i2#affiche()
method deplace p = i1#deplace p; i2#deplace p
end;;

```

On reprend alors le même principe que pour la classe *e_image*.

```

# class virtual ext_image () =
  object
  inherit image ()
  method virtual surface : unit -> float
  end;;
# class ext_rectangle x y w h =
  object
  inherit ext_image ()
  inherit rectangle x y w h
  method surface () = w *. h
  end;;
# class ext_cercle x y r=
  object
  inherit ext_image ()
  inherit cercle x y r
  method surface () = 3.14 *. r *.r
  end;;

```

L'extension de la classe *groupe* devient alors

```

# class ['a] ext_groupe eil ei2 =
  object
  inherit image()
  inherit ['a] groupe eil ei2
  method surface () = eil#surface() +. ei2#surface ()
  end;;

```

On obtient le programme suivant qui construit une liste le de type *ext_image*.

```

# let er = new ext_rectangle 1. 1. 2. 4. ;;
val er : ext_rectangle = <obj>
# let ec = new ext_cercle 1. 1. 8.;;
val ec : ext_cercle = <obj>
# let eg = new ext_groupe er ec;;

```

```

val eg : ext_rectangle ext_groupe = <obj>
# let le = [ (er:>ext_image); (ec :> ext_image); (eg :>
val le : ext_image list = [<obj>; <obj>; <obj>]
# List.map (fun x -> x#surface()) le;;
- : float list = [8; 200.96; 208.96]

```

Généralisation

Pour généraliser l'extension des traitements il est préférable d'intégrer des fonctions dans une méthode traitement et de construire une classe paramétrée par le type du résultat du traitement. Pour cela on définit la classe suivante :

```

# class virtual ['a] get_image (f: 'b -> unit -> 'a) =
  object (self:'b)
  inherit image ()
  method traitement () = f(self) ()
  end;;

```

Les classes suivantes possèdent alors un paramètre fonctionnel supplémentaire pour la construction de leurs instances.

```

# class ['a] get_rectangle f x y w h =
  object (self:'b)
  inherit ['a] get_image f
  inherit rectangle x y w h
  method get = (x,y,w,h)
  end;;
# class ['a] get_cercle f x y r=
  object (self:'b)
  inherit ['a] get_image f
  inherit cercle x y r
  method get = (x,y,r)
  end;;

```

L'extension de la classe *groupe* prend alors deux paramètres de type :

```

# class ['a,'c] get_groupe f etil eti2 =
  object
  inherit ['a] get_image f

```

```

inherit ['c] groupe etil eti2
method get = (i1,i2)
end;;

```

On obtient le programme qui étend le traitement des instances de *get_image*.

```

# let etr = new get_rectangle
  (fun r () -> let (x,y,w,h) = r#get in w *. h) 1. 1. 2.
val etr : float get_rectangle = <obj>
# let etc = new get_cercle
  (fun c () -> let (x,y,r) = c#get in 3.14 *. r *. r) 1.
val etc : float get_cercle = <obj>
# let etg = new get_groupe
  (fun g () -> let (i1,i2) = g#get in i1#traitement() +.
    (etr :> float get_image) (etc :> float get_image));;
val etg : (float, float get_image) get_groupe = <obj>
# let gel = [ (etr :> float get_image) ; (etc :> float get_image) ;
  (etg :> float get_image) ];;
val gel : float get_image list = [<obj>; <obj>; <obj>]
# List.map (fun x -> x#traitement()) gel;;
- : float list = [8; 200.96; 208.96]

```

L'extension des données et des traitements est plus facile dans le modèle objet en s'appuyant sur le modèle fonctionnel.

Organisations mixtes

Le dernier exemple de la section précédente a montré l'intérêt qu'il y avait à mixer les deux modèles pour le problème de l'extensibilité des composants. On propose maintenant de mixer modules paramétrés et liaison retardée pour profiter de la puissance de ces deux traits. L'application du foncteur produira de nouveaux modules contenant des classes utilisant le type et les fonctions du module paramètre. Si, de surcroît, la signature obtenue est compatible avec la signature du module paramètre, il est alors possible de réappliquer le module paramétré sur le module résultat, permettant ainsi de construire automatiquement de nouvelles classes.

Un exemple complet est donné dans la dernière partie de cet ouvrage qui est consacrée aux programmes concurrents et/ou distribués (page ??). Nous utilisons un foncteur pour engendrer un protocole de communication à partir d'un type donné; un second foncteur permet ensuite de déduire de ce protocole une classe implantant un serveur générique traitant des requêtes exprimées dans ce protocole. L'héritage peut ensuite être utilisé pour spécialiser le serveur vers ce qui est réellement le service attendu.

Exercices

Classes et modules pour structures de données

On désire construire des hiérarchies de classes à partir de l'application de foncteurs pour des structures de données classiques.

On définit les signatures suivantes :

```
# module type ELEMENT =  
  sig  
    class element :  
      string ->  
    object  
      method to_string : unit -> string  
      method of_string : string -> unit  
    end  
  end ;;
```

```
# module type STRUCTURE =  
  sig  
    class ['a] structure :  
    object  
      method add : 'a -> unit  
      method del : 'a -> unit  
      method mem : 'a -> bool  
      method get : unit -> 'a  
      method all : unit -> 'a list
```

```

method iter : ('a -> unit) -> unit
end
end ;;

```

1. Écrire un **module** à 2 paramètres M1 et M2 de types ELEMENT et STRUCTURE, construisant une sous-classe de *[a] structure* dont le 'a est contraint à *M1.element*.

```

# module Struct_Elmt (E:ELEMENT) (S:STRUCTURE) =
  struct
    class e_structure =
      object
        inherit [E.element] S.structure
      end
    end ;;
module Struct_Elmt :
  functor(E : ELEMENT) ->
  functor(S : STRUCTURE) ->
  sig
    class e_structure :
      object
        method add : E.element -> unit
        method all : unit -> E.element list
        method del : E.element -> unit
        method get : unit -> E.element
        method iter : (E.element -> unit) -> unit
        method mem : E.element -> bool
      end
    end
  end

```

2. Écrire un module simple **Entier** respectant la signature ELEMENT.

```

# module Entier : ELEMENT =
  struct
    class element v =
      object

```

```

val mutable n = int_of_string v
method to_string () = string_of_int n
method of_string x = n <- (int_of_string x)
end
end ;;
module Entier : ELEMENT

```

3. Écrire un module simple [Pile](#) respectant la signature de STRUCTURE.

```

# module Pile : STRUCTURE =
  struct
    class ['a] structure =
      object
        val mutable p = ( [] : 'a list )
        method add x = p <- x::p
        method del x = p <- List.filter ((<>) x) p
        method mem x = List.mem x p
        method get () = List.hd p
        method all () = p
        method iter f = List.iter f p
      end
    end ;;
module Pile : STRUCTURE

```

4. [Appliquer](#) le foncteur à ces deux paramètres.

```

# module Pile_Entier = Struct_Elmt(Entier)(Pile) ;;
module Pile_Entier :
  sig
    class e_structure :
      object
        method add : Entier.element -> unit
        method all : unit -> Entier.element list
        method del : Entier.element -> unit
        method get : unit -> Entier.element
        method iter : (Entier.element -> unit) -> unit
      end
  end

```

```

method mem : Entier.element -> bool
end
end

```

5. Modifier le **foncteur** initial en ajoutant les méthodes `to_string` et `of_string`.

```

# let split c s =
  let suffix s i =
    try String.sub s i ((String.length s)-i)
    with Invalid_argument("String.sub") -> ""
  in
  let rec split_from n =
    try let p = String.index_from s n c
    in (String.sub s n (p-n)) :: (split_from (p+1))
    with Not_found -> [ suffix s n ]
  in
  if s="" then [] else split_from 0 ;;
val split : char -> string -> string list = <fun>

# module Elmt_Struct (E:ELEMENT) (S:STRUCTURE) =
  struct
    class element v =
      object (self)
        val n = new S.structure
        method to_string () =
          let res = ref "" in
          let f x = res := !res ^ ((x#to_string ()) ^ " ")
          n#iter f ;
          !res
        method of_string x =
          let l = split ' ' x in
          List.iter (fun x -> n#add (new E.element x)) l
          initializer self#of_string v
        end
      end ;;
module Elmt_Struct :
  functor (E : ELEMENT) ->

```

```

functor(S : STRUCTURE) ->
sig
class element :
string ->
object
val n : E.element S.structure
method of_string : string -> unit
method to_string : unit -> string
end
end

```

6. Appliquer le foncteur de **nouveau**, puis l'appliquer au **résultat**.

Types abstraits

En reprenant l'exercice précédent, on cherche à implanter un module de signature ELEMENT dont la classe element utilise une variable d'instance d'un type abstrait.

On définit le type paramétré suivant :

```
# type 'a t = {mutable x : 'a t; f : 'a t -> unit};;
```

1. Écrire les **fonctions** apply, from_string et to_string. Ces deux dernières fonctions utilisent le module Marshal.

```

# let apply val_abs = val_abs.f val_abs.x ;;
val apply : 'a t -> unit = <fun>
# let from_string s = Marshal.from_string s 0 ;;
val from_string : string -> 'a = <fun>
# let to_string v = Marshal.to_string v [Marshal.Cl
val to_string : 'a -> string = <fun>

```

2. Écrire une **signature** S correspondant à la signature inférée précédemment en abstrayant le type t.


```

# module type S =
  sig
    type t
    val apply : t -> unit
    val from_string : string -> t
    val to_string : t -> string
  end ;;
module type S =
  sig
    type t
    val apply : t -> unit
    val from_string : string -> t
    val to_string : t -> string
  end

```

3. Écrire un **foncteur** qui prend un paramètre de signature S et retourne un module dont la signature est compatible avec ELEMENT.

```

# module Element_of (M:S) =
  struct
    class element v =
      object
        val mutable n = M.from_string v
        method to_string () = M.to_string n
        method of_string x = n <- M.from_string x
      end
    end ;;
module Element_of :
  functor(M : S) ->
  sig
    class element :
      string ->
      object
        val mutable n : M.t
        method of_string : string -> unit

```

```

method to_string : unit -> string
end
end

```

4. **Utiliser** le module résultat comme paramètre du module de l'exercice précédent.

```

# module Abstrait_Pile (M:S) = Elmt_Struct (Element
module Abstrait_Pile :
  functor (M : S) ->
  functor (S : STRUCTURE) ->
  sig
    class element :
      string ->
      object
        val n : Element_of(M).element S.structure
        method of_string : string -> unit
        method to_string : unit -> string
      end
    end
  end
end

```

Résumé

Ce chapitre a comparé les mérites respectifs des modèles d'organisation fonctionnel/modulaire et objet. Chacun essaie de répondre à sa manière aux difficultés de réutilisabilité et de modifiabilité du logiciel. Les différences principales proviennent de leur système de types, égalité de types entre paramètres de foncteurs et sous-typage en objet, et de l'évaluation avec la liaison tardive des objets. Les deux modèles n'arrivent pas à résoudre seuls le problème de l'extensibilité des composants, d'où la proposition d'une organisation mixte. Ce mélange d'organisation permet d'autre part de nouvelles structurations.

Pour en savoir plus

Le modèle modulaire souffre d'une faible réutilisation de code et de

difficultés pour la programmation incrémentielle. L'article < > ([AC96]) montre une extension simple du langage de modules, autorisant l'extension d'un module ainsi que la surcharge. Le choix du code d'un foncteur surchargé reprend les techniques de fonctions génériques de CLOS. La correction du système de types de ces modules étendus n'est pas établie.

La problématique du mélange des modèles est bien argumentée dans l'article < > ([FF98]), dans le sens de la facilité de réutilisation du code. Le problème de l'extensibilité des composants est largement décrit. Cet article est en version HTML à l'adresse suivante :

Lien: <http://www.cs.rice.edu/CS/PLT/Publications/icfp98-ff/paper.shtml>

On s'aperçoit dans ces propositions qu'il reste une part de typage dynamique pour la contrainte de type et/ou la résolution de conflits de types. Il n'est probablement pas déraisonnable d'ouvrir le typage statique pour obtenir des langages < > typés statiquement dans le but d'augmenter la réutilisabilité du code en autorisant un développement incrémentiel de celui-ci.

Applications

Introduction

Les deux applications de ce chapitre illustrent la structuration de programmes selon le modèle modulaire pour la première et objet pour la deuxième.

La première application propose un ensemble de modules paramétrés pour les jeux à deux joueurs. Un premier foncteur implante l'algorithme minimax-ab pour l'élagage de l'arbre de recherche. Un deuxième foncteur permet de changer l'interface homme-machine du jeu. Ces modules paramétrés sont ensuite appliqués à deux jeux : un morpion gravitationnel et un jeu de construction de lignes de force.

La deuxième application construit un monde où évoluent des robots. Le monde et les robots sont structurés en classes. Les différents comportements des robots sont obtenus par héritage d'une classe abstraite commune. Il est donc facile de définir de nouveaux comportements. Là aussi l'interface homme-machine peut être modifiée.

Chacune de ces deux applications, de part sa structuration, contient des composants qui sont réutilisables. Il est aisé de construire un nouveau jeu à deux joueurs, de même le mécanisme général de déplacement d'un robot dans un monde reste valable pour d'éventuels nouveaux robots.

Jeux à deux joueurs

L'application présentée dans cette section poursuit un double objectif. D'une part elle s'attaque à une résolution de problèmes dont la complexité empêche une exploration systématique de toutes les solutions, mettant ainsi en valeur la bonne adéquation du langage Objective CAML pour le traitement d'applications symboliques. D'autre part, elle permet grâce à l'utilisation de modules paramétrés, de définir un cadre général pour la construction de jeux à deux joueurs, permettant ainsi de factoriser une partie de la recherche et de rendre facilement modifiable des composants comme la fonction d'évaluation d'une position d'un jeu ou son affichage.

On présente tout d'abord la problématique des jeux à deux joueurs, puis nous décrivons l'algorithme *minimax-ab* permettant un élagage rapide de l'arbre des coups possibles. Nous [réalisons un module paramétré implantant cet algorithme pour décrire ensuite le module paramétré général des applications jeux à 2 joueurs. Enfin, nous appliquons ces foncteurs pour réaliser deux applications jeux : un puissance 4 (un morpion gravitationnel), et Stone Henge (un jeu de construction de lignes de force).

Problématique des jeux à 2 joueurs

Les jeux à deux joueurs font partie des applications classiques en

programmation symbolique. C'est un bon exemple de résolution de problèmes pour au moins deux raisons :

- le nombre de solutions à analyser pour obtenir le meilleur coup nécessite des méthodes autres que la force brute : aux échecs le nombre de coups possibles est en moyenne de 30 et une partie se joue en une quarantaine de coups pour chaque joueur; ce qui donne quelques 30^{80} positions pour explorer l'arbre complet d'une partie!
- la qualité d'une solution est facilement vérifiable, en particulier il est possible de tester la qualité de la solution proposée par un programme en utilisant un autre programme.

Supposons que l'on puisse utiliser une exploration totale de toutes les parties possibles à partir d'une position légale du jeu. Un tel programme a besoin d'une fonction de génération des coups légaux à partir de cette position et d'une fonction d'évaluation donnant un score à la position donnée. La fonction d'évaluation donne un score maximal à une position gagnante et un score minimal à une position perdante. À partir de la position initiale, on peut donc construire l'arbre de toutes les variantes de la partie, où chaque noeud correspond à une position, ses fils aux positions suivantes obtenues en ayant joué un coup et les feuilles aux positions gagnantes, perdantes ou nulles. Une fois cet arbre construit, son exploration permet de déterminer s'il existe un chemin menant à la victoire, ou à une position nulle le cas échéant. Le chemin de plus petite longueur peut alors être choisi pour amener au résultat voulu.

Comme la taille d'un tel arbre est en règle générale trop grande pour être envisageable, il est nécessaire d'en limiter sa construction. La première possibilité est de limiter la profondeur de recherche, c'est-à-dire le nombre de coups et de réponses à évaluer. On réduit ainsi la profondeur de l'arbre donc sa taille. Dans ce cas on atteint rarement des feuilles à moins d'être en fin de partie.

D'autre part, nous pouvons essayer de limiter le nombre de coups engendrés pour pouvoir les évaluer plus finement. Pour cela, nous tentons de n'engendrer que les coups semblant les plus favorables et de les examiner en commençant par les meilleurs. Cela permet ainsi d'élaguer rapidement des branches entières de l'arbre. C'est justement ce que propose l'algorithme *minimax-ab* qui est présenté

dans la section suivante.

Minimax

On présente la recherche *minimax* puis on décrit sa variante optimisée par les coupures ab. L'implantation de cet algorithme utilise un module FAlphabeta paramétré par la représentation du jeu et la fonction d'évaluation. On distingue les deux joueurs en les nommant A et B.

Minimax

L'algorithme *minimax* est un algorithme de recherche en profondeur, avec une profondeur limitée. Il nécessite d'utiliser :

- une fonction de génération des coups légaux à partir d'une position,
- et une fonction d'évaluation d'une position de jeu.

À partir d'une position du jeu, l'algorithme explore l'arbre de tous les coups légaux jusqu'à la profondeur demandée. Les scores des feuilles de l'arbre sont alors calculés par la fonction d'évaluation. Un score positif indique une bonne position pour le joueur A, un score négatif une mauvaise position pour le joueur A donc une bonne position pour le joueur B. Selon le joueur qui joue, le passage d'une position à une autre est maximisante (pour le joueur A) ou minimisante (pour le joueur B). Les joueurs essaient de jouer les coups les plus profitables pour eux-mêmes. En cherchant le meilleur coup pour le joueur A, la recherche en profondeur 1 cherchera à déterminer le coup immédiat qui maximise le score de la nouvelle position.

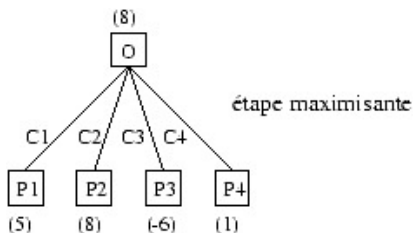


Figure 17.1 : recherche maximisante à un niveau

Sur la figure 17.1, le joueur A part de la position O, détermine quatre coups légaux, construit ces nouvelles configurations et les évalue. De ces scores, sa meilleure position est P2 (de score 8). Il propage cette valeur à la position O, indiquant par là que cette position amène en un coup à une nouvelle position de score 8 en jouant le coup C2. L'exploration en profondeur 1 est en règle générale insuffisante, car on ne tient pas compte de la réponse de l'adversaire. Cela produit des programmes cherchant le gain immédiat de matériel (comme la prise d'une reine aux échecs), sans s'apercevoir que les pièces sont protégées ou que la position devient perdante (gambit de la reine pour faire mat). Une exploration de profondeur 2 permet de s'apercevoir du contrecoup.

La figure 17.2 montre un niveau supplémentaire de développement de l'arbre en tenant compte de la réponse du joueur B. Celui-ci recherche aussi son meilleur coup. Pour cela l'algorithme *minimax* minimisera les scores des noeuds de profondeur 2.

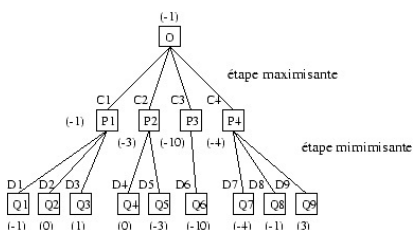


Figure 17.2 : recherche maximisante et minimisante à deux niveaux

Le coup P2 qui amenait à une position immédiate de score 8, va en fait amener la position du jeu à un score de -3. En effet si B joue le coup D5, alors le score de la position Q5 vaut -3. On s'aperçoit alors que le coup C1 limite les dégâts avec un score de -1. Il sera donc préféré.

Dans la plupart des jeux, il est possible de faire lanterner son adversaire, en le faisant jouer à coups forcés, dans le but d'embrouiller la situation en espérant qu'il commette une faute. Pour cela la recherche de profondeur 2 est très insuffisante pour le côté tactique du jeu. Le côté stratégique est rarement bien exploité par un programme car il n'a pas la vision de la probable évolution de la position en fin de partie. La difficulté de profondeur plus grande provient de l'explosion combinatoire. Par exemple aux échecs, l'exploration de 2 profondeurs supplémentaires apporte un facteur d'environ mille fois plus de combinaisons ($30 * 30$). Donc si on cherche à calculer une profondeur de 10, on obtiendra environ 5^{14} positions, ce qui est bien entendu trop. Pour cela on essaie d'élaguer l'arbre de recherche. On peut noter que dans la figure 17.2, il n'est pas forcément utile d'explorer la branche P3 dans la mesure où le score de cette position à la profondeur 1 (voir figure 17.1) est déjà moins bon que celui trouvé dans la branche P1. De même la branche P4 n'a pas besoin d'être complètement explorée. Dès le calcul de Q7, on obtient un score inférieur à celui de P1 (toujours complètement explorée). Les calculs de Q8 et Q9 ne pourront pas améliorer cette situation même si leur score respectif est meilleur que Q7. Dans une étape minimisante, le score le plus faible est remonté. On sait déjà qu'ils n'apporteront rien de nouveau. La variante ab du minimax utilise cet élagage pour diminuer le nombre de branches à explorer.

Minimax-ab

On appelle coupure a la borne inférieure d'un noeud maximisant, et coupure b la borne supérieure d'un noeud minimisant. La figure

17.3 montre les coupures effectuées dans les branches P3 et P4 par la connaissance de la borne inférieure -1 de P1.

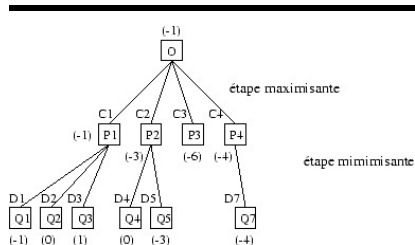


Figure 17.3 : coupure a dans une étape max-min

Dès que l'arbre est plus large ou plus profond le nombre de coupures augmente, élagant ainsi de grands sous-arbres.

Module paramétré pour le minimax-ab

On désire réaliser un module paramétré FAlphabeta qui implante cet algorithme, réutilisable pour tout jeu à deux joueurs. Ses paramètres correspondent d'une part à toutes les informations d'arbitrage du jeu et d'autre part à la fonction d'évaluation.

Interfaces

On déclare les signatures REPRESENTATION, pour la représentation du jeu, et EVAL, pour l'évaluation d'une position.

```
# module type REPRESENTATION = sig
  type jeu
  type coup
  val jeu_depart : unit -> jeu
  val coups_legaux: bool -> jeu -> coup list
  val jouer: bool -> coup -> jeu -> jeu
end;;
```

```

# module type EVAL =
  sig
    type jeu
    val evaluer: bool -> jeu -> int
    val plusI : int
    val moinsI: int
    val est_feuille: bool -> jeu -> bool
    val est_stable: bool -> jeu -> bool
    type etat = G | P | N | C
    val etat_de : bool -> jeu -> etat
  end;;

```

Les types *jeu* et *coup* sont abstraits. Un joueur sera représenté par un booléen. La fonction `coups_legaux` prend un joueur, une position et retourne la liste des coups possibles. La fonction `jouer` prend un joueur, un coup, une position et retourne une nouvelle position. Les valeurs `plusI` et `moinsI` sont les bornes des valeurs retournées par la fonction `evaluer`. Le prédicat `est_feuille` vérifie si un joueur dans une position donnée peut jouer. Le prédicat `est_stable` indique si la position pour le joueur est dans une situation stable. Les résultats des ces fonctions influencent la poursuite de l'exploration de coups quand on atteint la profondeur prévue.

La signature ALPHABETA correspond à la signature résultant de l'application totale du module paramétré que l'on désire réaliser. Celle-ci masque les différentes fonctions auxiliaires que nous utiliserons pour l'implantation de l'algorithme.

```

# module type ALPHABETA = sig
  type jeu
  type coup
  val alphabeta : int -> bool -> jeu -> coup
end ;;

```

La fonction `alphabeta` prend en paramètre la profondeur de la recherche, le joueur et la position du jeu, elle retourne le coup déterminé.

On définit enfin la signature fonctionnelle FALPHABETA que devra

respecter l'implantation du foncteur.

```
# module type FALPHABETA =  
  functor (Rep : REPRESENTATION) ->  
  functor (Eval : EVAL with type jeu = Rep.jeu) ->  
  ALPHABETA with type jeu = Rep.jeu and type coup = Rep.coup
```

Implantation

Le module paramétré FAlphabetO explicite le partage du type *jeu* entre ses deux paramètres Rep et Eval. Ce module ne comprend que six fonctions et deux exceptions. Le joueur true cherche à maximiser son score alors que le joueur false le minimise. La fonction *maxmin_iter* calcule le maximum entre le meilleur score des branches à partir d'un coup du joueur true et la borne a. La fonction *maxmin* prend quatre paramètres : *prof* qui indique la profondeur actuelle du calcul, *noeud* une position du jeu, *alpha* et *beta* les bornes des coupures. Si le noeud est une feuille de l'arbre ou si la profondeur maximale est atteinte alors la fonction retourne l'évaluation de la position. Si ce n'est pas le cas, elle applique à tous les coups légaux du joueur true la fonction *maxmin_iter* en lui passant la fonction de poursuite de calcul dont la profondeur est décrémentée (minmax). Cette dernière cherchera à minimiser le score de la réponse du joueur false. Les coupures sont implantées par des exceptions. Si la coupure b est déclenchée dans l'itération sur les coups légaux de la fonction *maxmin*, alors celle-ci retourne immédiatement avec la valeur propagée par l'exception. Les fonctions *minmax_iter* et *minmax* effectuent le travail pour l'autre joueur. La fonction *cherche* détermine le coup à jouer à partir du meilleur score en parcourant les listes des scores et des coups. La fonction principale *alphabet* de ce module calcule les coups légaux d'une position pour un joueur donné et lance les recherches à la profondeur indiquée puis retourne le meilleur coup.

```
# module FAlphabetO  
  (Rep : REPRESENTATION) (Eval : EVAL with type jeu = Rep.jeu)  
  struct  
    type jeu = Rep.jeu  
    type coup = Rep.coup
```

```
exception AlphaCoupure of int
```

```
exception BetaCoupure of int
```

```
let maxmin_iter noeud minmax_cur beta alpha cp =
```

```
let alpha_resu =
```

```
max alpha (minmax_cur (Rep.jouer true cp noeud) beta al
```

```
in if alpha_resu >= beta then raise (BetaCoupure alpha)
```

```
else alpha_resu
```

```
let minmax_iter noeud maxmin_cur alpha beta cp =
```

```
let beta_resu =
```

```
min beta (maxmin_cur (Rep.jouer false cp noeud) alpha b
```

```
in if beta_resu <= alpha then raise (AlphaCoupure beta)
```

```
else beta_resu
```

```
let rec maxmin prof noeud alpha beta =
```

```
if (prof < 1 & Eval.est_stable true noeud)
```

```
or Eval.est_feuille true noeud
```

```
then Eval.evaluer true noeud
```

```
else
```

```
try let prev = maxmin_iter noeud (minmax (prof - 1)) be
```

```
in List.fold_left prev alpha (Rep.coups_legaux true noe
```

```
with BetaCoupure a -> a
```

```
and minmax prof noeud beta alpha =
```

```
if (prof < 1 & Eval.est_stable false noeud)
```

```
or Eval.est_feuille false noeud
```

```
then Eval.evaluer false noeud
```

```
else
```

```
try let prev = minmax_iter noeud (maxmin (prof - 1)) al
```

```
in List.fold_left prev beta (Rep.coups_legaux false noe
```

```
with AlphaCoupure b -> b
```

```
let rec cherche a l1 l2 = match (l1,l2) with
```

```
(h1::q1, h2::q2) -> if a = h1 then h2 else cherche a q1
```

```
| ([], []) -> failwith ("AB: "^(string_of_int a)^" non
```

```
| (_, _) -> failwith "AB: longueur diff"
```

```
(* val alphabeta : int -> bool -> Rep.jeu -> Rep.coup *)
```

```
let alphabeta prof joueur racine =
```

```

let alpha = ref Eval.moinsI and beta = ref Eval.plusI i
let l = ref [] in
let cpl = Rep.coups_legaux joueur racine in
let eval =
try
for i = 0 to (List.length cpl) - 1 do
if joueur then
let b = Rep.jouer joueur (List.nth cpl i) racine in
let a = minmax (prof-1) b !beta !alpha
in l := a :: !l ;
alpha := max !alpha a ;
(if !alpha >= !beta then raise (BetaCoupure !alpha) )
else
let a = Rep.jouer joueur (List.nth cpl i) racine in
let b = maxmin (prof-1) a !alpha !beta
in l := b :: !l ;
beta := min !beta b ;
(if !beta <= !alpha then raise (AlphaCoupure !beta))
done ;
if joueur then !alpha else !beta
with
BetaCoupure a -> a
| AlphaCoupure b -> b
in
l := List.rev !l ;
cherche eval !l cpl
end ;;

```

On peut alors fermer le module FAlphabetaO en lui associant cette signature.

```

# module FAlphabeta = (FAlphabetaO : FALPHABETA);;
module FAlphabeta : FALPHABETA

```

C'est ce dernier module qui est ensuite utilisé sur différentes représentations et fonctions d'évaluation de jeu.

Organisation d'un programme de jeux

L'organisation d'un programme de jeu à 2 joueurs doit bien séparer

la partie spécifique à un jeu donné de la partie commune à toutes les applications de jeux. Pour cela on se propose d'utiliser plusieurs modules paramétrés par les modules spécifiques, permettant ainsi d'éviter de réécrire à chaque fois les parties communes. La figure 17.4 montre l'organisation choisie.

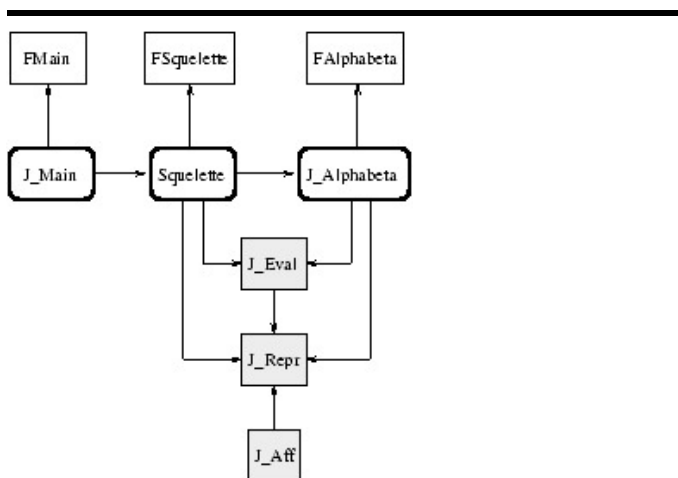


Figure 17.4 : organisation d'une application de jeux

Les modules sur fond blanc correspondent aux parties communes de l'application. Ce sont des modules paramétrés. On y retrouve le foncteur FAlphabeta décrit précédemment. Les modules sur fond grisé sont les modules spécifiques au jeu. Les trois principaux sont la représentation du jeu (J_Repr), l'affichage du jeu (J_Aff) et la fonction d'évaluation (J_Eval). Les modules aux bords gras arrondis sont obtenus par l'application des modules paramétrés sur les modules simples spécifiques au jeu.

Le module FAlphabeta vient d'être décrit. Les deux autres modules communs sont FMain qui contient la boucle principale et FSquelette pour la gestion des joueurs.

Module FMain

Le module Fmain contient le point d'entrée de l'exécution d'un programme de jeux. Il est paramétré par un module de signature SQUELETTE de description de l'interaction avec un joueur dont voici la définition.

```
# module type SQUELETTE = sig
  val accueil: unit -> unit
  val init: unit -> ((unit -> unit) * (unit -> unit))
  val encore: unit -> bool
  val fin: unit -> unit
  exception Gagne
  exception Perd
  exception Nul
  val gagne: unit -> unit
  val perd: unit -> unit
  val nul: unit -> unit
end ;;
```

La fonction init construit un couple des fonctions d'action de chaque joueur. Les autres fonctions servent à gérer l'interaction d'une partie. Le module FMain contient deux fonctions : ca_joue qui fait alterner les joueurs et main qui gère la boucle principale.

```
# module FMain (P : SQUELETTE) =
  struct
    let ca_joue tours = while true do (fst tours) () ; (snd tours) ()

    let main () = let fini = ref false
    in P.accueil ();
    while not !fini do
      ( try ca_joue (P.init ())
      with P.Gagne -> P.gagne ()
      | P.Perd -> P.perd ()
      | P.Nul -> P.nul () );
      fini := not (P.encore ())
    done ;
    P.fin ()
  end ;;

module FMain :
  functor(P : SQUELETTE) ->
```

```

sig
val ca_joue : (unit -> 'a) * (unit -> 'b) -> unit
val main : unit -> unit
end

```

Module FSquelette

Le module paramétré FSquelette gère les tours de chaque joueur selon les indications fournies en début de partie comme la nature des joueurs (programme ou non) et l'ordre des joueurs. Il a besoin de plusieurs paramètres pour la représentation du jeu, son affichage, sa fonction d'évaluation et la recherche ab comme cela est décrit à la figure 17.4.

On donne d'abord la signature manquante pour l'affichage.

```

# module type AFFICHAGE = sig
  type jeu
  type coup
  val accueil: unit -> unit
  val fin: unit -> unit
  val gagne: unit -> unit
  val perd: unit -> unit
  val nul: unit -> unit
  val init: unit -> unit
  val affiche : bool -> coup -> jeu -> jeu -> unit
  val choix : bool -> jeu -> coup
  val q_jouer : unit -> bool
  val q_commencer : unit -> bool
  val q_continuer : unit -> bool
end ;;

```

Il est à noter que la représentation du jeu et celle des coups doivent être partagées par tous les modules paramètres, d'où les contraintes sur les types. Les deux fonctions principales sont tourH et tourM, elles gèrent respectivement le tour d'un joueur humain (utilisant la fonction Aff.choix) et celui d'un programme. La fonction init détermine la nature des joueurs suite aux réponses des appels à Aff.q_jouer


```

# module FSquelette
(Rep    : REPRESENTATION)
(Aff     : AFFICHAGE with type jeu = Rep.jeu and type cou
(Eval    : EVAL with type jeu = Rep.jeu)
(Alpha   : ALPHABETA with type jeu = Rep.jeu and type cou
struct
let prof = ref 4
exception Gagne
exception Perd
exception Nul
let gagne = Aff.gagne
let perd = Aff.perd
let nul = Aff.nul
let encore = Aff.q_continuer
let le_jeu = ref (Rep.jeu_depart())
let fin = Aff.fin
let accueil = Aff.accueil

let tourH joueur () =
let choix = Aff.choix joueur !le_jeu in
let ancien_jeu = !le_jeu
in le_jeu := Rep.jouer joueur choix !le_jeu ;
Aff.affiche joueur choix ancien_jeu !le_jeu ;
match Eval.etat_de joueur !le_jeu with
Eval.P -> raise Perd
| Eval.G -> raise Gagne
| Eval.N -> raise Nul
| _      -> ()

let tourM joueur () =
let choix = Alpha.alphabeta !prof joueur !le_jeu in
let ancien_jeu = !le_jeu
in le_jeu := Rep.jouer joueur choix !le_jeu ;
Aff.affiche joueur choix ancien_jeu !le_jeu ;
match Eval.etat_de joueur !le_jeu with
Eval.G -> raise Gagne
| Eval.P -> raise Perd
| Eval.N -> raise Nul
| _      -> ()

```

```

let init () =
let a = Aff.q_jouer () in
let b = Aff.q_jouer()
in le_jeu :=Rep.jeu_depart () ;
Aff.init () ;
match (a,b) with
true,true -> tourM true, tourM false
| true,false -> tourM true, tourH false
| false,true -> tourH true, tourM false
| false,false -> tourH true, tourH false
end ;;

```

Les parties indépendantes du jeu sont maintenant réalisées. On peut donc commencer la programmation de différents jeux. Cette organisation modulaire facilite les modifications de l'affichage ou de la fonction d'évaluation d'un jeu comme nous allons le voir.

Puissance 4

On s'intéresse tout d'abord à un premier jeu simple, un morpion gravitationnel, appelé habituellement Puissance 4. Le jeu est représenté par sept colonnes de six lignes. À chaque tour un joueur pose dans une colonne un jeton de sa couleur, celui-ci tombe jusqu'au premier emplacement libre de sa colonne. Si une colonne est complètement remplie, aucun joueur ne peut y jouer. La partie se termine quand un des joueurs a relié une ligne de quatre pièces (horizontale, verticale, ou diagonale), dans ce cas il a gagné, ou quand toutes les colonnes sont pleines et dans ce cas la partie est nulle. La figure 17.5 montre une partie finie.

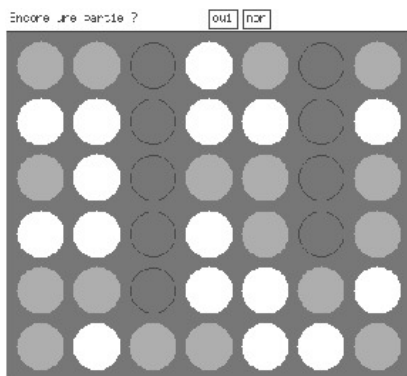


Figure 17.5 : Une partie de Puissance 4

On remarque une ligne de quatre jetons gris en diagonale partant du coin en bas à droite.

Représentation du jeu : module P4_rep

On choisit pour ce jeu une représentation matricielle. Chaque case de la matrice est vide ou bien contient un jeton d'un des deux joueurs. Un coup est le numéro d'une colonne. Les coups légaux sont les colonnes dont la dernière case n'est pas remplie.

```
# module P4_rep = struct
  type cell = A | B | Vide
  type jeu = cell array array
  type coup = int
  let col = 7 and lig = 6
  let jeu_depart () = Array.create_matrix lig col Vide

  let coups_legaux b m =
    let l = ref [] in
    for c = 0 to col-1 do if m.(lig-1).(c) = Vide then l :=
      !l

  let ajoute mat c =
```

```

let l = ref lig
in while !l > 0 & mat.(!l-1).(c-1) = Vide do decr l don

let jouer_gen cp m e =
let mj = Array.map Array.copy m
in mj.((ajoute mj cp)-1).(cp-1) <- e ; mj

let jouer b cp m = if b then jouer_gen cp m A else joue
end ;;

```

On peut facilement vérifier que ce module accepte les contraintes de la signature REPRESENTATION.

```

# module P4_rep_T = (P4_rep : REPRESENTATION) ;;
module P4_rep_T : REPRESENTATION

```

Affichage du jeu : module P4_text

Le module P4_text décrit une interface textuelle pour le jeu Puissance 4 compatible avec la signature AFFICHAGE. Il ne présente pas de grande difficulté, mais permet de bien comprendre ensuite l'assemblage des modules.

```

# module P4_text = struct
  open P4_rep
  type jeu = P4_rep.jeu
  type coup = P4_rep.coup

  let print_jeu mat =
    for l = lig - 1 downto 0 do
      for c = 0 to col - 1 do
        match mat.(l).(c) with
        A   -> print_string "X "
        | B   -> print_string "O "
        | Vide -> print_string ". "
      done;
    print_newline ()
  done ;
  print_newline ()

```

```

let accueil () = print_string "P4 ...\n"
let fin () = print_string "A bientôt ... \n"
let question s = print_string s; print_string " o/n ? "
let q_commencer () = question "Voulez-vous commencer"
let q_continuer () = question "Encore une partie"
let q_jouer () = question "Est-ce une machine qui joue"

let gagne ()= print_string "Le 1er joueur gagne" ; print_string "\n"
let perd () = print_string "Le 1er joueur perd" ; print_string "\n"
let nul () = print_string "Partie nulle" ; print_newline 1

let init () = print_string "X: 1er joueur   O: 2eme joueur\n"
print_newline () ; print_newline () ;
print_jeu (jeu_depart ()) ; print_newline()

let affiche b c aj j = print_jeu j

let est_coup = function '1'..'7' -> true | _ -> false

exception Coup of int
let rec choix joueur jeu =
  print_string ("Choix joueur" ^ (if joueur then "1" else "2")) ;
  let l = coups_legaux joueur jeu
  in try while true do
    let i = read_line()
    in ( if (String.length i > 0) && (est_coup i.[0])
    then let c = (int_of_char i.[0]) - (int_of_char '0')
    in if List.mem c l then raise (Coup c) );
    print_string "coup non valable, essayez encore : "
  done ;
  List.hd l
  with Coup i -> i
  | _ -> List.hd l
end ;;

```

On vérifie immédiatement qu'il respecte bien les contraintes de la signature AFFICHAGE.

```

# module P4_text_T = (P4_text : AFFICHAGE) ;;

```

```
module P4_text_T : AFFICHAGE
```

Fonction d'évaluation : module P4_eval

La qualité d'un jeu dépend principalement de la fonction d'évaluation d'une position. Le module P4_eval définit la fonction `evaluer` qui calcule la valeur d'une position pour un joueur donné. Cette fonction appelle la fonction `eval_bloc`, dans les quatre directions en spécialisant les diagonales, qui elle-même appelle la fonction `eval_quatre` pour calculer le nombre de jetons dans la ligne demandée. Le tableau `valeur` donne la valeur d'un bloc contenant 0, 1, 2 ou 3 jetons de la même couleur. L'exception `Quatre` est déclenchée quand 4 jetons sont alignés.

```
# module P4_eval = struct
  open P4_rep
  type jeu = P4_rep.jeu
  let valeur = Array.of_list [0; 2; 10; 50]
  exception Quatre of int
  exception Valeur_nulle
  exception Arg_invalid
  let moinsI = -10000
  let plusI = 10000

  let eval_quatre m l_dep c_dep delta_l delta_c =
    let n = ref 0 and e = ref Vide
    and x = ref c_dep and y = ref l_dep
    in try
      for i = 1 to 4 do
        if !y < 0 or !y >= lig or !x < 0 or !x >= col then raise Arg_invalid
        ( match m.(!y).(!x) with
          A    -> if !e = B then raise Valeur_nulle ;
          incr n ;
          if !n = 4 then raise (Quatre plusI) ;
          e := A
          | B    -> if !e = A then raise Valeur_nulle ;
          incr n ;
          if !n = 4 then raise (Quatre moinsI);
          e := B;
```

```

| Vide -> ()
x := !x + delta_c ;
y := !y + delta_l
done ;
valeur.(!n) * (if !e=A then 1 else -1)
with
Valeur_nulle | Arg_invalid -> 0

let eval_bloc m e cmin cmax lmin lmax dx dy =
for c=cmin to cmax do for l=lmin to lmax do
e := !e + eval_quatre m l c dx dy
done done

let evaluer b m =
try let evaluation = ref 0
in (* evaluation des lignes *)
eval_bloc m evaluation 0 (lig-1) 0 (col-4) 0 1 ;
(* evaluation des colonnes *)
eval_bloc m evaluation 0 (col-1) 0 (lig-4) 1 0 ;
(* diagonales partant de la premiere ligne (à droite) *)
eval_bloc m evaluation 0 (col-4) 0 (lig-4) 1 1 ;
(* diagonales partant de la premiere colonne (à droite) *)
eval_bloc m evaluation 1 (lig-4) 0 (col-4) 1 1 ;
(* diagonales partant de la premiere ligne (à gauche) *)
eval_bloc m evaluation 3 (col-1) 0 (lig-4) 1 (-1) ;
(* diagonales partant de la derniere colonne (à gauche) *)
eval_bloc m evaluation 1 (lig-4) 3 (col-1) 1 (-1) ;
!evaluation
with Quatre v -> v

let est_feuille b m = let v = evaluer b m
in v=plusI or v=moinsI or coups_legaux b m = []

let est_stable b j = true

type etat = G | P | N | C

let etat_de_joueur m =
let v = evaluer_joueur m
in if v = plusI then if joueur then G else P

```

```

else if v = moinsI then if joueur then P else G
else if coups_legaux joueur m = [] then N else C
end ;;

```

Le module P4_eval est compatible avec les contraintes de la signature EVAL

```

# module P4_eval_T = (P4_eval : EVAL) ;;
module P4_eval_T : EVAL

```

Pour faire jouer deux fonctions d'évaluation l'une contre l'autre, il faut modifier la fonction evaluer en aiguillant chaque joueur sur sa fonction d'évaluation propre.

Assemblage des modules

Toutes les briques nécessaires à la réalisation du jeu puissance 4 sont maintenant réalisées. Il ne reste plus qu'à les assembler selon le schéma de la figure 17.4. On construit tout d'abord le squelette P4_Squelette qui est l'application du module paramétré FSquelette aux modules P4_rep, P4_text, P4_eval et au résultat de l'application du module paramétré FAlphabeta sur P4_rep et P4_eval.

```

# module P4_squelette =
  FSquelette (P4_rep) (P4_text) (P4_eval) (FAlphabeta (P4_rep))
module P4_squelette :
  sig
    val prof : int ref
    exception Gagne
    exception Perd
    exception Nul
    val gagne : unit -> unit
    val perd : unit -> unit
    val nul : unit -> unit
    val encore : unit -> bool
    val le_jeu : P4_text.jeu ref
    val fin : unit -> unit
    val accueil : unit -> unit
    val tourH : bool -> unit -> unit

```



```

val tourM : bool -> unit -> unit
val init : unit -> (unit -> unit) * (unit -> unit)
end

```

On obtient alors le module principal P4_main de l'application du module paramétré FMain sur le résultat de l'application précédente P4_squelette.

```

# module P4_main = FMain(P4_squelette) ;;
module P4_main :
  sig
    val ca_joue : (unit -> 'a) * (unit -> 'b) -> unit
    val main : unit -> unit
  end

```

Le lancement du jeu s'effectue par l'application de la fonction P4_main.main sur ().

Test du jeu

Tel que le squelette général des jeux a été écrit, il est possible de faire jouer deux personnes entre elles (le programme ne sert que d'arbitre), une personne contre le programme ou le programme contre lui-même. Bien que cette dernière possibilité ne présente que peu d'intérêt pour un joueur, elle permet de lancer des tests plus facilement sans devoir attendre la réponse d'un joueur. Le jeu suivant choisit cette option.

```

# P4_main.main ();;
P4 ...
Est-ce une machine qui joue ? o/n ? o
Est-ce une machine qui joue ? o/n ? o
X: 1er joueur    O: 2eme joueur

```

```

. . . . .
. . . . .
. . . . .
. . . . .

```

```

. . . . .
. . . . .

```

Une fois la position initiale affichée, le joueur 1 (joué par le programme) calcule son coup qui est ensuite affiché.

```

. . . . .
. . . . .
. . . . .
. . . . .
. . . . .
. . . . . X .

```

Le joueur 2 (toujours joué par le programme) calcule son propre coup et le jeu s'affiche. Le jeu se poursuit ainsi de suite jusqu'à une position de fin de partie. Dans le cas présent le joueur 1 gagne la partie sur la position finale suivante :

```

. O O O . O .
. X X X . X .
X O O X . O .
X X X O . X .
X O O X X O .
X O O O X X O

```

```

Le 1er joueur gagne
Encore une partie o/n ? n
A bientot ...
- : unit = ()

```

Interface graphique

Comme le plaisir de jouer provient aussi de l'interface graphique du programme, on définit un nouveau module P4_graph compatible avec la signature AFFICHAGE qui ouvre une fenêtre graphique et gère les clics souris. On retrouve le texte de ce module dans le sous-catalogue Applications du CD-ROM (voir page ??).

On crée alors un nouveau squelette (P4_squeletteG) qui résulte de l'application du module paramétré FSquelette.

```
# module P4_squeletteG =
  FSquelette (P4_rep) (P4_graph) (P4_eval) (FAlphabeta (F
```

Seul le paramètre du module d'affichage diffère de la version texte dans l'application du module paramétré FSquelette. On crée alors le module principal de l'application Puissance 4 avec interface graphique.

```
# module P4_mainG = FMain(P4_squeletteG) ;;
module P4_mainG :
  sig
    val ca_joue : (unit -> 'a) * (unit -> 'b) -> unit
    val main : unit -> unit
  end
```

L'évaluation de l'expression P4.mainG.main() affiche la fenêtre graphique de la figure 17.5 et gère l'interaction avec le joueur.

Stone Henge

Le jeu Stone Henge, dû à Reiner Knizia, est un jeu de construction de lignes de force. Les règles sont simples à apprendre, mais l'intérêt du jeu reste grand même après de nombreuses parties. La règle en ligne peut être consultée au lien suivant :

Lien: <http://www.gamecabinet.com/frenchRules/Stonehenge.html>

La position initiale du jeu est représentée à la figure 17.6.

Présentation du jeu

Le but du jeu est de gagner au moins 8 lignes de force (lignes claires) sur les 15 existantes. Le gain d'une ligne est visualisé par la pose d'un menhir (ou mégalithe) sur une case gris foncé associée à une ligne de force.

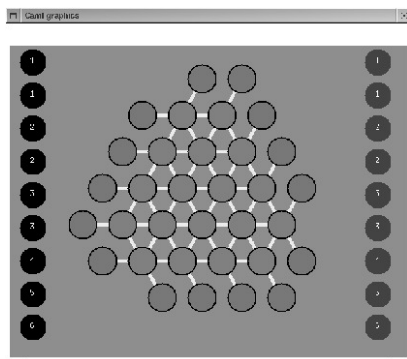


Figure 17.6 : Position initiale de Stone Henge

À son tour chaque joueur pose une pierre (sur les 9 qu'il possède au début) numérotée (de 1 à 6) sur une des 18 cases centrales grises. On ne peut pas poser une pierre sur une case occupée. Une fois une pierre posée, une ou plusieurs lignes de force peuvent être perdues ou gagnées. Une ligne de force est gagnée par un joueur si l'autre joueur ne peut pas dépasser le score (total des pierres déjà posées d'une couleur) du premier, même en remplissant les cases encore vides de cette ligne par ses meilleures pierres. Par exemple, à la figure 17.7, le joueur noir commence par poser la pierre de valeur 3, le joueur rouge sa pierre 2, puis le joueur noir joue sa pierre 6 et gagne une ligne. Le joueur rouge pose alors sa pierre 4 et gagne aussi une ligne. Celle-ci n'est pas encore complètement remplie, mais le joueur noir ne pourra jamais dépasser le score de son adversaire. On peut noter que le joueur rouge aurait pu jouer une pierre 3 à la place de sa pierre 4. En effet il reste une seule case libre sur cette ligne de force et la pierre la plus forte de noir vaut 5, donc noir ne peut plus battre rouge sur cette ligne.

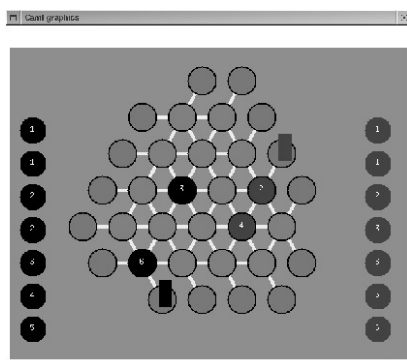


Figure 17.7 : Position après 4 coups

Dans un cas d'égalité sur une ligne pleine, celui qui a posé la dernière pierre et qui n'a pas pu battre le score de son adversaire perd cette ligne. La figure 17.8 montre un tel cas. Le dernier coup de rouge est la pierre 4. Sur la ligne pleine où le 4 est posé il y a égalité. Comme rouge est le dernier joueur à y avoir posé une pierre, il n'a pas pu battre son adversaire et perd la ligne, d'où le menhir noir qui indique le gain de noir. On s'aperçoit que la fonction jouer qui effectue le rôle d'arbitre doit tenir compte de ces subtilités de pose de menhirs.

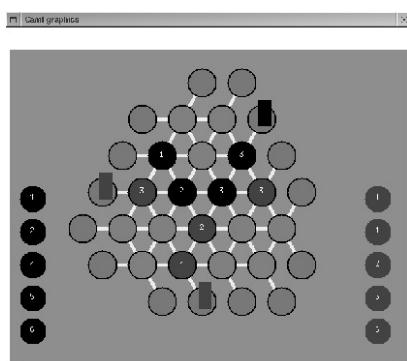


Figure 17.8 : Position après 6 coups

Il n'y a jamais de partie nulle à ce jeu. Il y a 15 lignes et toutes sont prises à un moment de la partie, donc un des deux joueurs atteint le gain d'au moins 8 lignes.

Complexité de la recherche

Avant toute implantation d'un nouveau jeu, il est nécessaire de calculer le nombre de coups légaux entre deux tours de jeu ainsi que le nombre de coups d'une partie. On détermine alors la profondeur raisonnable de l'algorithme minimax-ab. Dans le jeu StoneHenge le nombre de coups d'une partie est borné par le nombre de jetons des deux joueurs, c'est-à-dire 18. Le nombre de coups possibles diminue au fur et à mesure de la partie. Au premier coup de la partie, le joueur a 6 jetons différents et 18 cases libres, ce qui lui fait 108 coups légaux. Au deuxième coup, le deuxième joueur a aussi 6 jetons différents pour 17 cases libre (102 coups légaux). Le fait de passer d'une profondeur de 2 à 4 pour les premiers coups de la partie fait passer d'environ 10^4 coups à 10^8 coups. Par contre en fin de partie, disons sur les 8 derniers coups, la complexité se réduit fortement. Si on calcule le pire des cas (tous les jetons sont différents) on obtient environ 23 millions de possibilités :

$$4*8 * 4*7 * 3*6 * 3*5 * 2*4 * 2*3 * 1*2 * 1*1 = 23224320$$

Il semble délicat de calculer au delà d'une profondeur 2 pour les premiers coups. Cela dépend de la fonction d'évaluation, et de sa capacité à évaluer les positions de début de partie, du moins tant qu'il y a peu de menhirs posés. Par contre en fin de partie la profondeur peut augmenter à 4 ou 6, mais cela est probablement trop tard pour récupérer une position affaiblie.

Implantation

La réalisation de ce jeu suit l'organisation utilisée pour le jeu Puissance 4 et décrite à la figure 17.4. Les deux principales

difficultés viennent du respect des règles du jeu pour la pose des menhirs et de la fonction d'évaluation qui doit pouvoir évaluer une situation le plus rapidement possible tout en étant pertinente.

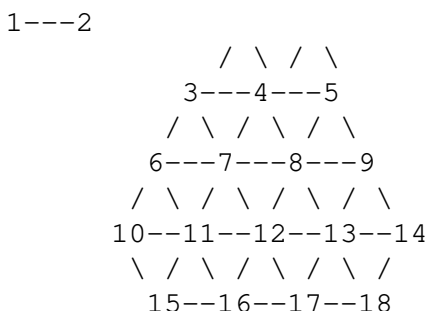
On décrit tout d'abord la représentation du jeu et l'arbitre de jeu, pour s'intéresser par la suite à la fonction d'évaluation.

Représentation du jeu

Il y a quatre données particulières pour ce jeu :

- les pierres (ou jetons) des joueurs (type *pierre*)
- les menhirs (type *menhir*)
- les 15 lignes de force
- les 18 cases où poser les pierres

On donne un numéro unique à chaque case.



Par chaque case passe 3 lignes de force. On numérote aussi chaque ligne de force. Cette description se retrouve dans la déclaration de la liste lignes, que l'on convertit en vecteur (vecteur_l). Une case est soit vide, soit contient la pierre posée et son propriétaire. On conserve d'autre part pour chaque case le numéro des lignes qui y passent. Ce tableau est calculé par la fonction lignes_par_case et est nommé num_ligne_par_case.

Le jeu est représenté par le vecteur des 18 cases, le vecteur des 15 lignes de force contenant ou non un menhir, et des listes de jetons pour les deux joueurs. La fonction jeu_depart crée ces quatre éléments. Le calcul des coups légaux pour un joueur revient à un

produit cartésien des jetons restants avec les cases libres. Plusieurs fonctions utilitaires permettent de compter le score d'un joueur dans une ligne, de calculer le nombre de cases libres d'une ligne et de vérifier si une ligne a déjà un menhir. Il ne reste plus qu'à écrire la fonction jouer qui joue un coup et décide des menhirs à poser. Nous la décrivons à la fin du listing suivant du module Stone_rep.

```
# module Stone_rep = struct
  type joueur = bool
  type pierre = P of int
  let int_of_pierre = function P x -> x

  type menhir = Rien | M of joueur

  type case = Vide | Occup of joueur*pierre
  let value_on_case = function
    Vide -> 0
    | Occup (j, x) -> int_of_pierre x

  type jeu = J of case array * menhir array * pierre list
  type coup = int * pierre

  let lignes = [
    [0;1]; [2;3;4]; [5; 6; 7; 8;]; [9; 10; 11; 12; 13]; [14
    [0; 2; 5; 9]; [1; 3; 6; 10; 14]; [4; 7; 11; 15]; [8; 12
    [9; 14]; [5; 10; 15]; [2; 6; 11; 16]; [0; 3; 7; 12; 17]

  let vecteur_l = Array.of_list lignes

  let lignes_par_case v =
  let t = Array.length v in
  let r = Array.create 18 [|] in
  for i = 0 to 17 do
    let w = Array.create 3 0
    and p = ref 0 in
    for j=0 to t-1 do if List.mem i v.(j) then (w.(!p) <- j
  done;
  r.(i) <- w
done;
r
```



```

let num_ligne_par_case = lignes_par_case vecteur_l
let rec lignes_de_i i l = List.filter (fun t -> List.mem t l) l

let lignes_des_cases l =
let a = Array.create 18 l in
for i=0 to 17 do
a.(i) <- (lignes_de_i i l)
done; a

let ldc = lignes_des_cases lignes

let jeu_depart () = let lp = [6; 5; 4; 3; 3; 2; 2; 1; 1]
J ( Array.create 18 Vide, Array.create 15 Rien,
List.map (fun x -> P x) lp, List.map (fun x -> P x) lp)

let rec unicite l = match l with
[] -> []
| h::t -> if List.mem h t then unicite t else h:: (unicite t)

let coups_legaux joueur (J (ca, m, r1, r2)) =
let r = if joueur then r1 else r2 in
if r = [] then []
else
let l = ref [] in
for i = 0 to 17 do
if value_on_case ca.(i) = 0 then l:= i:: !l
done;
let l2 = List.map (fun x->
List.map (fun y-> x,y) (List.rev(unicite r)) ) !l in
List.flatten l2
let copie_plateau p = Array.copy p

let copie_carnac m = Array.copy m
let rec joue_pierre caillou l = match l with
[] -> []
| x::q -> if x=caillou then q
else x::(joue_pierre caillou q)

let compte_case joueur case = match case with

```

```

Vide -> 0
| Occup (j,p) -> if j = joueur then (int_of_pierre p) e

let compte_ligne joueur ligne pos =
List.fold_left (fun x y -> x +  compte_case joueur pos.

let rec compte_max n = function
[] -> 0
| t::q ->
if (n>0) then
(int_of_pierre t) + compte_max (n-1) q
else
0

let rec nbr_cases_libres ca l = match l with
[] -> 0
| t::q -> let c = ca.(t) in
match c with
Vide -> 1 + nbr_cases_libres ca

```

Des robots en goguette

L'exemple de ce paragraphe illustre l'utilisation d'objets ainsi que la bibliothèque graphique. Nous verrons comment Objective CAML reprend les notions d'héritage simple, d'héritage multiple, de redéfinition de méthode et de liaison dynamique. Nous verrons également comment les classes paramétrées peuvent être mises à profit.

L'application comprend deux catégories principales d'objets : un monde et des robots. Le monde est un ensemble de cases sur lesquelles évoluent des robots. Nous aurons plusieurs classes de robots. Chacune d'elles possédera sa propre stratégie de déplacement dans le monde. Le principe d'interaction du monde et des robots est ici extrêmement simple. Le monde est entièrement maître du jeu : il demande tour à tour à chacun des robots qu'il connaît quelle est sa prochaine position. Chaque robot détermine sa prochaine position à l'aveugle : il ne connaît ni la géométrie du monde, ni les autres robots présents. Si la position demandée par un robot est légale et libre alors le monde l'y déplace.

Le monde matérialisera l'évolution des robots par une interface. La complexité (toute relative) de la conception et du développement de cet exemple est dans la toujours nécessaire séparation entre un traitement (ici : l'évolution des robots) et son interface (ici : la trace de cette évolution).

Description générale.

L'application est développée en deux temps.

1. un ensemble de définitions donnant des classes de calcul pur pour le monde et pour divers robots envisagés.
2. un ensemble de définitions, utilisant les précédentes et ajoutant ce qui est nécessaire à la mise en place d'une interface. Nous donnerons deux exemples d'interfaces : une rudimentaire sous forme de texte ; une, plus élaborée, utilisant la bibliothèque graphique.

Robots « éthérés »

Dans un premier temps, nous nous intéressons aux robots hors de toute considération sur l'environnement qui les entourent, c'est à dire de l'interface qui les affiche.

```
# class virtual robot (i0:int) (j0:int) =  
  object  
    val mutable i = i0  
    val mutable j = j0  
    method get_pos = (i,j)  
    method set_pos (i', j') = i <- i'; j <- j'  
    method virtual next_pos : unit -> (int * int)  
  end ;;
```

En toute généralité, un robot est une entité connaissant, ou croyant connaître, sa position (i et j), capable de la donner à qui la lui demande (get_pos), susceptible de modifier cette connaissance si on la lui précise (set_pos) et sachant décider d'un éventuel mouvement

vers une nouvelle position (next_pos).

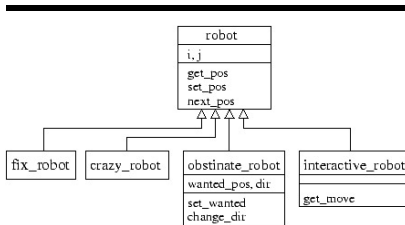


Figure 17.9 : hiérarchie de classes des robots purs

Pour améliorer la lisibilité du programme, nous définissons les mouvements relatifs à une direction absolue :

```
# type dir = North | East | South | West | Nothing ;;
# let walk (x,y) = function
  North -> (x,y+1) | South -> (x,y-1)
  | West -> (x-1,y) | East -> (x+1,y)
  | Nothing -> (x,y) ;;
val walk : int * int -> dir -> int * int = <fun>
# let turn_right = function
  North -> East | East -> South | South -> West | West -> North
val turn_right : dir -> dir = <fun>
```

Du schéma induit par la classe virtuelle des robots, nous définissons quatre espèces de robots distinctes (voir la figure 17.9) en précisant leur manière de se déplacer :

- les robots fixes qui ne bougent jamais :

```
# class fix_robot i0 j0 =
  object
  inherit robot i0 j0
  method next_pos() = (i,j)
  end ;;
```

- les robots fous qui se déplacent au hasard :

```
# class crazy_robot i0 j0 =
  object
  inherit robot i0 j0
  method next_pos () = ( i+(Random.int 3)-1 , j+(Random.int 3)-1 )
  end ;;
```

- les robots obstinés qui conservent la même direction tant qu'ils peuvent avancer,

```
# class obstinate_robot i0 j0 =
  object(self)
  inherit robot i0 j0
  val mutable wanted_pos = (i0,j0)
  val mutable dir = West

  method private set_wanted_pos d = wanted_pos <- wanted_pos
  method private change_dir = dir <- turn_right dir
  method next_pos () = if (i,j) = wanted_pos
  then let np = walk (i,j) dir in ( wanted_pos <- np
  else ( self#change_dir ; wanted_pos <- (i,j) ; (i,j)
  end ;;
```

- les robots téléguidés qui obéissent à un opérateur extérieur :

```
# class virtual interactive_robot i0 j0 =
  object(self)
  inherit robot i0 j0
  method virtual private get_move : unit -> dir
  method next_pos () = walk (i,j) (self#get_move ())
  end ;;
```

Le cas du robot interactif est différent des autres car son comportement est lié à l'interface qui permettra de lui communiquer des ordres. En attendant, nous nous appuyons sur une méthode qui, virtuellement, communique cet ordre et en conséquence la classe `interactive_robot` demeure abstraite.

Remarquons que non seulement les quatre classes de robots spécialisés héritent de la classe robot mais que de surcroît elles en ont le type. En effet, les seules méthodes que nous ayons ajoutées sont des méthodes privées et donc n'apparaissent pas dans le type des instances de ces classes (voir page ??). Ce point nous est indispensable si nous souhaitons considérer tous les robots comme des objets de même type.

Monde pur

Un monde *pur* est un monde indépendant de l'interface. Il y est connu l'ensemble des positions qu'un robot est susceptible d'occuper. Cela prend la forme d'une grille de taille $l \times h$, d'une méthode `is_legal` assurant qu'un couple d'entiers est bien une position dans le monde, et d'une méthode `is_free` indiquant si un robot occupe ou non une position donnée.

En outre, un monde dispose de la liste des robots présents sur sa surface robots ainsi que d'une méthode `add` permettant de faire rentrer de nouveaux robots.

Pour finir, un monde est pourvu de la méthode `run` lui permettant de prendre vie.

```
# class virtual ['robot_type] world (l0:int) (h0:int) =  
  object (self)  
    val l = l0  
    val h = h0  
    val mutable robots = ( [] : 'robot_type list )  
    method add r = robots <- r::robots  
    method is_free p = List.for_all (fun r -> r#get_pos <>  
    method virtual is_legal : (int * int) -> bool  
  
    method private run_robot r =  
      let p = r#next_pos ()  
      in if (self#is_legal p) & (self#is_free p) then r#set_p  
  
    method run () =  
      while true do List.iter (function r -> self#run_robot r  
      end ;;
```

```

class virtual ['a] world :
  int ->
  int ->
  object
  constraint 'a =
  < get_pos : int * int; next_pos : unit -> int * int;
  set_pos : int * int -> unit; .. >
  val h : int
  val l : int
  val mutable robots : 'a list
  method add : 'a -> unit
  method is_free : int * int -> bool
  method virtual is_legal : int * int -> bool
  method run : unit -> unit
  method private run_robot : 'a -> unit
end

```

Le système de type d'Objective CAML ne permet pas de laisser le type des robots non déterminé (voir page ??). Pour résoudre ce problème, nous avons la possibilité de restreindre ce type à celui de la classe robot. Mais dans ce cas, nous nous interdisions de pouvoir peupler un monde d'autres objets que ceux ayant exactement le même type que robot. Donc, nous avons choisi de paramétrer la classe world par le type des robots qui le peuplent. Nous pourrions ensuite instancier ce paramètre de type par des robots textuels ou par des robots graphiques.

Robots textuels

Des objets texte

Pour obtenir des robots gérables par une interface texte, nous définissons la classe des objets textuels (txt_object).

```

# class txt_object (s0:string) =
  object

```

```

val name = s0
method get_name = name
end ;;

```

Une classe de spécification : les robots textuels abstraits

Par héritage double de robots et txt_object, nous obtenons la classe abstraite txt_robot des robots textuels.

```

# class virtual txt_robot i0 j0 =
  object
    inherit robot i0 j0
    inherit txt_object "Anonymous"
  end ;;
class virtual txt_robot :
  int ->
  int ->
  object
    val mutable i : int
    val mutable j : int
    val name : string
    method get_name : string
    method get_pos : int * int
    method virtual next_pos : unit -> int * int
    method set_pos : int * int -> unit
  end

```

Cette classe nous sert pour définir un monde à interface texte (voir page ??). Les habitants de ce monde ne seront ni des objets de txt_robot (puisque cette classe est abstraite) ni des héritiers de cette classe. La classe txt_robots est en quelque sorte une classe de spécification permettant au compilateur d'identifier les types des méthodes (calcul et interface) des habitants du monde à interface texte. L'utilisation d'une telle classe de spécification vient de la séparation que nous voulons maintenir entre les calculs et l'interface.

Les robots concrets en mode texte

Ils s'obtiennent simplement par double héritage; la figure 17.10 donne leur hiérarchie de classes.

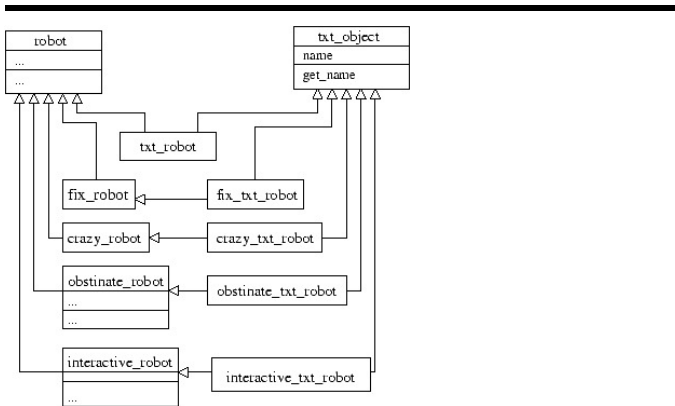


Figure 17.10 : hiérarchie de classes des robots en mode texte

```
# class fix_txt_robot i0 j0 =  
  object  
  inherit fix_robot i0 j0  
  inherit txt_object "Fix robot"  
end ;;  
  
# class crazy_txt_robot i0 j0 =  
  object  
  inherit crazy_robot i0 j0  
  inherit txt_object "Crazy robot"  
end ;;  
  
# class obstinate_txt_robot i0 j0 =  
  object  
  inherit obstinate_robot i0 j0  
  inherit txt_object "Obstinate robot"  
end ;;
```

Les robots interactifs doivent pour devenir concrets définir leur

méthode d'interaction avec l'utilisateur.

```
# class interactive_txt_robot i0 j0 =  
  object  
    inherit interactive_robot i0 j0  
    inherit txt_object "Interactive robot"  
    method private get_move () =  
      print_string "Which dir : (n)orth (e)ast (s)outh (w)est"  
    match read_line() with  
      "n" -> North   | "s" -> South  
      | "e" -> East   | "w" -> West  
      | _  -> Nothing  
    end ;;
```

Monde textuel

Le monde à interface texte se dérive du monde pur par

1. héritage de la classe générique world en instanciant son paramètre de type par la classe de spécification txt_robot,
2. redéfinition de la méthode run pour y inclure les différents affichages textuels.

```
# class virtual txt_world (l0:int) (h0:int) =  
  object (self)  
    inherit [txt_robot] world l0 h0 as super  
  
    method private display_robot_pos r =  
      let (i,j) = r#get_pos in Printf.printf "(%d,%d)" i j  
  
    method private run_robot r =  
      let p = r#next_pos ()  
      in if (self#is_legal p) & (self#is_free p)  
      then  
        begin  
          Printf.printf "%s is moving from " r#get_name ;  
          self#display_robot_pos r ;  
          print_string " to " ;
```

```

r#set_pos p;
self#display_robot_pos r ;
end
else
begin
Printf.printf "%s is staying at " r#get_name ;
self#display_robot_pos r
end ;
print_newline () ;
print_string"next - ";
ignore (read_line())

method run () =
let print_robot r =
Printf.printf "%s is at " r#get_name ;
self#display_robot_pos r ;
print_newline ()
in
print_string "Initial state :\n";
List.iter print_robot robots;
print_string "Running :\n";
super#run() (* 1 *)
end ;;

```

Nous attirons l'attention du lecteur sur l'appel à la méthode `run` de la classe ancêtre (marqué `(* 1 *)` dans le code) dans la redéfinition de cette même méthode. Nous avons là une illustration des deux types de liaison des méthodes possibles : statique ou dynamique (voir page ??). L'appel à `super#run` est statique; c'est l'intérêt de nommer la superclasse que de pouvoir appeler ses méthodes alors qu'elles ont été redéfinies. Par contre, dans cette méthode `super#run` se trouve un appel à `self#run_robot`. C'est ici une liaison dynamique qui a lieu; c'est la méthode définie dans la classe `txt_world` qui est exécutée et non celle de `world`, sans quoi nous n'obtiendrions aucun affichage.

Le monde plan rectangulaire textuel

s'obtient en implantant la dernière méthode encore abstraite :

is_legal.

```
# class closed_txt_world l0 h0 =  
  object (self)  
  inherit txt_world l0 h0  
  method is_legal (i,j) = (0<=i) & (i<l) & (0<=j) & (j<h)  
  end ;;
```

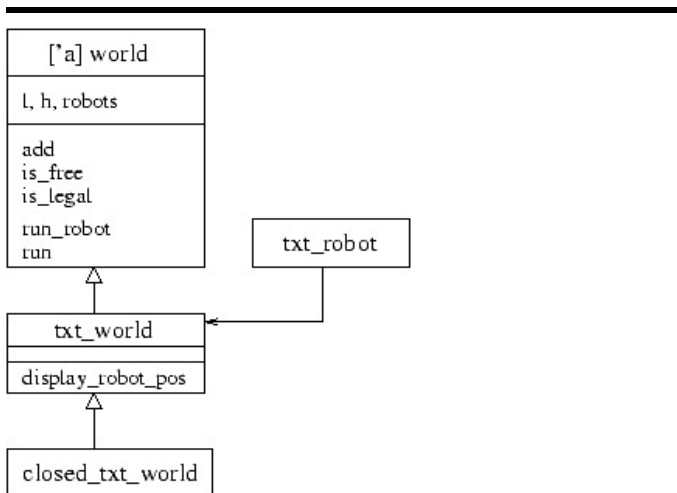


Figure 17.11 : hiérarchie de classes du monde plan rectangulaire en mode texte

On peut procéder à un petit essai en tapant :

```
let w = new closed_txt_world 5 5  
and r1 = new fix_txt_robot 3 3  
and r2 = new crazy_txt_robot 2 2  
and r3 = new obstinate_txt_robot 1 1  
and r4 = new interactive_txt_robot 0 0  
in w#add r1; w#add r2; w#add r3; w#add r4; w#run () ;;
```

Nous allons passer à présent à la réalisation de l'interface graphique pour notre monde de robots. En fin de course, nous obtiendrons une

application ayant l'apparence de la figure [17.12](#).

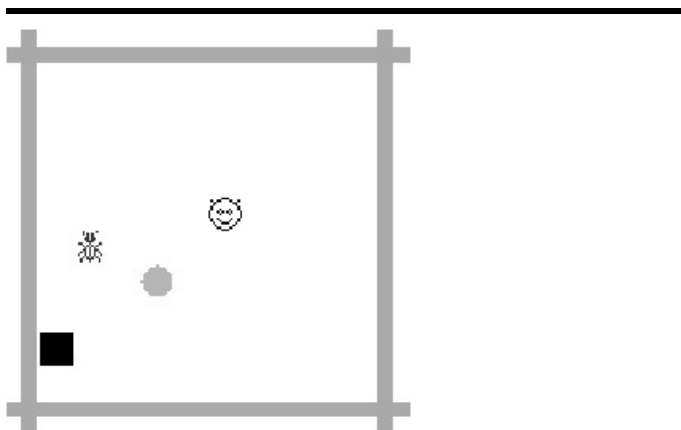


Figure 17.12 : Le monde graphique des robots

Robots graphiques

Nous obtenons des robots en mode graphique en suivant le même schéma que le mode texte :

1. définition d'un objet graphique générique,
2. définition d'une classe abstraite de robots graphiques par double héritage des robots et des objets graphiques (analogue de la classe de spécification du paragraphe [17](#)),
3. définition par double héritage des robots possédant un comportement particulier.

Objets graphiques génériques

Un objet graphique simple est un objet possédant une méthode `display` qui prend en argument les coordonnées d'un pixel et

s'affiche.

```
# class virtual graph_object =  
  object  
  method virtual display : int -> int -> unit  
  end ;;
```

De cette spécification, il est possible de tirer des objets graphiques extrêmement complexes. Nous allons nous contenter ici d'une classe `graph_item` affichant le bitmap qui sert à la construire.

```
# class graph_item x y im =  
  object (self)  
    val size_box_x = x  
    val size_box_y = y  
    val bitmap = im  
    val mutable last = None  
  
    method private erase = match last with  
      Some (x,y,img) -> Graphics.draw_image img x y  
      | None -> ()  
  
    method private draw i j = Graphics.draw_image bitmap i j  
    method private keep i j =  
      last <- Some (i,j,Graphics.get_image i j size_box_x size_box_y)  
  
    method display i j = match last with  
      Some (x,y,img) -> if x<>i || y<>j  
      then ( self#erase ; self#keep i j ; self#draw i j )  
      | None -> ( self#keep i j ; self#draw i j )  
    end ;;
```

Un objet `graph_item` conserve la portion d'image sur laquelle il est affiché pour la restaurer lors de l'affichage suivant. De plus, si l'image n'a pas bougé elle n'est pas réaffichée.

```
# let foo_bitmap = [| [| Graphics.black |] |] ;;  
# class square_item x col =  
  object  
    inherit graph_item x x (Graphics.make_image foo_bitmap)  
    method private draw i j = Graphics.set_color col ;
```

```

Graphics.fill_rect (i+1) (j+1) (x-2) (x-2)
end ;;

# class disk_item r col =
  object
  inherit graph_item (2*r) (2*r) (Graphics.make_image foo)
  method private draw i j = Graphics.set_color col ;
    Graphics.fill_circle (i+r) (j+r) (r-2)
  end ;;

# class file_bitmap_item name =
  let ch = open_in name
  in let x = Marshal.from_channel ch
  in let y = Marshal.from_channel ch
  in let im = Marshal.from_channel ch
  in let () = close_in ch
  in object
  inherit graph_item x y (Graphics.make_image im)
  end ;;

```

Nous avons spécialisé les `graph_item` en carrés, disques et bitmaps lus depuis un fichier.

Le robot graphique abstrait

est à la fois un robot et un objet graphique.

```

# class virtual graph_robot i0 j0 =
  object
  inherit robot i0 j0
  inherit graph_object
  end ;;

```

Les robots graphiques fixes, fous et obstinés

sont des objets graphiques spécialisés.

```

# class fix_graph_robot i0 j0 =

```

```

object
inherit fix_robot i0 j0
inherit disk_item 7 Graphics.green
end ;;

# class crazy_graph_robot i0 j0 =
object
inherit crazy_robot i0 j0
inherit file_bitmap_item "crazy_bitmap"
end ;;

# class obstinate_graph_robot i0 j0 =
object
inherit obstinate_robot i0 j0
inherit square_item 15 Graphics.black
end ;;

```

Le robot graphique interactif

utilise les primitives `key_pressed` et `read_key` du module `Graphics` pour l'acquisition du déplacement. On reconnaîtra les touches 8, 6, 2 et 4 du pavé numérique (touche `NumLock` active). De cette façon, l'utilisateur n'est pas obligé de donner une indication de déplacement à chaque interrogation du monde.

```

# class interactive_graph_robot i0 j0 =
object
inherit interactive_robot i0 j0
inherit file_bitmap_item "interactive_bitmap"
method private get_move () =
if not (Graphics.key_pressed ()) then Nothing
else match Graphics.read_key() with
  '8' -> North | '2' -> South | '4' -> West | '6' -> East
end ;;

```

Monde graphique

On obtient un monde à interface graphique par héritage du monde pur en instanciant le paramètre 'a_robot avec la classe abstraite des robots graphiques graph_robot. Comme pour le monde en mode texte, le monde graphique redéfinit la méthode run_robot de traitement d'un robot et la méthode d'activation générale run.

```
# let delay x = let t = Sys.time () in while (Sys.time () - t < x) do ()

# class virtual graph_world l0 h0 =
  object (self)
    inherit [graph_robot] world l0 h0 as super
    initializer
      let gl = (l+2)*15 and gh = (h+2)*15 and lw=7 and cw=7
      in Graphics.open_graph (" "^(string_of_int gl)^"x"^(string_of_int gh))
      Graphics.set_color (Graphics.rgb 170 170 170) ;
      Graphics.fill_rect 0 lw gl lw ;
      Graphics.fill_rect (gl-2*lw) 0 lw gh ;
      Graphics.fill_rect 0 (gh-2*cw) gl cw ;
      Graphics.fill_rect lw 0 lw gh

    method run_robot r = let p = r#next_pos ()
      in delay 0.001 ;
      if (self#is_legal p) & (self#is_free p)
      then ( r#set_pos p ; self#display_robot r)

    method display_robot r = let (i,j) = r#get_pos
      in r#display (i*15+15) (j*15+15)

    method run() = List.iter self#display_robot robots ;
      super#run()
    end ;;
```

Notez que la fenêtre graphique est créée à l'initialisation d'un objet de cette classe.

Le monde plan rectangulaire et graphique

s'obtient de la même manière que pour le monde plan rectangulaire et textuel.

```
# class closed_graph_world 10 h0 =
  object (self)
    inherit graph_world 10 h0
    method is_legal (i,j) = (0<=i) & (i<1) & (0<=j) & (j<h)
    end ;;
class closed_graph_world :
  int ->
  int ->
  object
    val h : int
    val l : int
    val mutable robots : graph_robot list
    method add : graph_robot -> unit
    method display_robot : graph_robot -> unit
    method is_free : int * int -> bool
    method is_legal : int * int -> bool
    method run : unit -> unit
    method run_robot : graph_robot -> unit
  end
```

On peut alors tester l'application graphique en tapant

```
let w = new closed_graph_world 10 10 ;;
w#add (new fix_graph_robot 3 3) ;;
w#add (new crazy_graph_robot 2 2) ;;
w#add (new obstinate_graph_robot 1 1) ;;
w#add (new interactive_graph_robot 5 5) ;;
w#run () ;;
```

Pour en faire plus

L'implantation de la méthode run_robot des différents mondes sous-entend que les robots sont potentiellement capables de se rendre en tout point du monde du moment que celui-ci est libre et légal. De plus, rien n'interdit à un robot de modifier sa position sans en prévenir le monde. Une amélioration possible consiste à faire gérer l'ensemble des positions des robots par le monde; lors du déplacement d'un robot, le monde vérifie d'une part si la nouvelle

position est légale mais aussi si elle constitue un déplacement autorisé. Dans ce cas, le robot devra être capable de demander au monde sa propre position; ce qui entraîne que la classe des robots devra être dépendante de la classe du monde. On pourra définir une classe robot prenant comme paramètre de type une classe de monde.

Cette modification permet alors de définir des robots capables d'interroger le monde qui les entoure et donc de se comporter en fonction de celui-ci. Nous pourrions réaliser des robots qui suivent ou qui fuient d'autres robots, qui tentent de les bloquer, etc. L'étape suivante est de permettre aux robots de communiquer entre eux pour s'échanger des informations et constituer ainsi des équipes de robots.

Les chapitres de la partie suivante de l'ouvrage permettent de libérer l'exécution des robots les uns des autres : soit en ayant recours aux Threads (voir page ??) pour que chacun s'exécute sur un processus distinct, soit en profitant des possibilités de l'informatique distribuée (voir page ??) pour que les robots soient des *clients* s'exécutant sur des machines distantes qui annoncent leur déplacement ou demandent des informations à un monde qui serait un *serveur*. Ce problème est traité à la page ??.

Part IV Concurrency et répartition

Présentation de la partie IV

Cette quatrième partie est une introduction aux concepts de la programmation parallèle et présente les modèles à mémoire partagée et à mémoire répartie. Il n'est pas nécessaire de posséder un super-calculateur parallèle pour pouvoir exprimer des algorithmes concurrents ou réaliser des applications réparties. Nous définissons dans ce préambule les différents termes utilisés dans les chapitres suivants.

Dans le cadre de la séquentialité, une instruction d'un programme s'exécute après une autre. On parle alors de dépendance causale. Ces programmes possèdent la propriété de déterminisme. Pour un même jeu de données en entrée, un même programme termine ou bien ne termine pas, et dans le cas où il termine, produit le même résultat. Le déterminisme indique qu'une cause a un et un seul effet. Les seules exceptions, déjà rencontrées en Objective CAML, proviennent des fonctions prenant en entrée une information extérieure au programme, comme la fonction `Sys.time`.

Dans le cadre de la programmation parallèle, un programme est découpé en plusieurs processus actifs. Chacun de ces processus est séquentiel, mais plusieurs instructions, appartenant aux différents processus, sont exécutées en parallèle, c'est-à-dire « en même temps ». La séquentialité se transforme en concurrence. On parle alors d'indépendance causale. Une cause peut avoir plusieurs effets, mutuellement exclusifs (un seul effet se produit). Une conséquence immédiate est de perdre la propriété de déterminisme : un même programme, pour une même donnée, termine ou ne termine pas en pouvant produire lorsqu'il termine des résultats différents.

Pour contrôler l'exécution d'un programme parallèle, il est nécessaire d'utiliser deux nouvelles notions :

- la synchronisation qui introduit une attente d'une condition sur plusieurs processus;
- la communication pour l'envoi d'informations d'un ou plusieurs processus à un ou plusieurs processus.

Du point de vue de la causalité, la synchronisation assure que plusieurs causes indépendantes doivent s'être produites avant que l'effet puisse avoir lieu. La communication possède une contrainte temporelle : un message ne peut pas être reçu avant d'avoir été émis. La communication peut s'effectuer selon plusieurs modes : communication d'un processus à un autre (point-à-point) ou en diffusion (un-à-tous ou tous-à-tous).

Les deux modèles de programmation parallèle, décrits à la figure 17.13, diffèrent sur le contrôle de l'exécution par les formes de synchronisation et de communication.

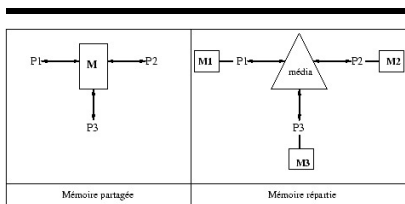


Figure 17.13 : Modèles de parallélisme

Chaque processus P_i correspond à un processus séquentiel. L'ensemble de ces processus, interagissant sur une mémoire commune (M), ou communiquant via un média, constitue une application parallèle.

Modèle à mémoire partagée

La communication dans le modèle à mémoire partagée est implicite.

L'information est transmise lors de l'écriture dans une zone de la mémoire partagée, et récupérée quand un autre processus vient lire cette zone. Par contre la synchronisation doit être explicite, en utilisant des instructions élémentaires de gestion de l'exclusion mutuelle et d'attente sur condition.

Ce modèle est utilisé dès que l'on se sert de manière concurrente une ressource commune. On peut citer en particulier la construction des systèmes d'exploitation.

Modèle à mémoire répartie

Dans ce modèle chaque processus séquentiel P_i possède sa propre mémoire privée M_i . Il est le seul à y avoir accès. Les processus doivent alors communiquer pour transférer de l'information à travers un média assurant ces transferts. La difficulté de ce modèle provient de l'implantation du média. Les programmes s'en chargeant s'appellent des *protocoles*.

Ceux-ci sont organisés en couches. Les protocoles de haut niveau, implantant des services élaborés, utilisent les couches de plus bas niveaux.

Il existe plusieurs types de communication selon la capacité du média à stocker de l'information et selon le caractère bloquant ou non de l'émission et de la réception. On parle de communications synchrones quand le transfert d'information n'est possible qu'après une synchronisation globale des processus émetteur et récepteur. Dans ce cas l'émission et la réception peuvent être bloquantes.

Si le média possède une capacité de stockage, il peut conserver des messages émis en vue de leur acheminement ultérieur. La communication peut être alors asynchrone et l'émission non bloquante. Il est toutefois nécessaire de spécifier la capacité de stockage du média, l'ordre d'acheminement, les délais et la fiabilité des transmissions.

Enfin si l'émission est non bloquante avec un média ne pouvant conserver les messages, on obtient une communication évanescente : seuls les processus récepteurs prêts reçoivent le

message émis qui est alors perdu pour les autres.

Dans le modèle à mémoire répartie la communication est explicite alors que la synchronisation est implicite (elle est en fait produite par la communication). C'est le dual du modèle à mémoire partagée.

Parallélisme physique et logique

Le modèle à mémoire répartie est valable dans le cas de parallélisme physique (réseau d'ordinateurs) ou logique (processus Unix communiquant par tubes de communication ou processus légers communiquant par canaux). Il n'y a pas de valeurs globales connues par tous les processus (comme une horloge globale).

Le modèle à mémoire partagée est plus proche du parallélisme physique, une même mémoire est effectivement partagée. Néanmoins, rien n'empêche de simuler une mémoire partagée sur un réseau d'ordinateurs.

Cette quatrième partie va donc montrer comment construire des applications parallèles en Objective CAML en utilisant les deux modèles présentés. Elle s'appuie principalement sur la bibliothèque Unix, qui interface les appels système Unix à Objective CAML, et la bibliothèque Thread qui implante les processus légers. La bibliothèque Unix est en grande partie portée sous Windows, en particulier les fonctions sur les descripteurs de fichier. Ceux-ci sont utilisés pour lire et écrire sur des fichiers, mais aussi pour les tubes de communication (pipe) et pour les prises de communication réseau (socket).

Le chapitre 18 décrit les traits essentiels de la bibliothèque Unix en s'intéressant tout particulièrement à la communication d'un processus vers l'extérieur ou vers d'autres processus. La notion de processus de ce chapitre est celui de « processus lourd » à la Unix. Leur création s'effectue par l'appel système fork qui duplique le contexte d'exécution et la zone mémoire des données en créant une filiation entre processus. L'interaction entre processus est réalisée soit par des signaux, soit à travers des tubes de communication.

Le chapitre 19 reprend la notion de processus en présentant les processus légers de la bibliothèque Thread. À la différence des processus lourds précédents, ceux-ci ne dupliquent que le contexte d'exécution d'un processus existant. La zone de données est donc commune au thread créateur et au thread créé. Selon le style de programmation, les processus légers d'Objective CAML permettent de rendre compte du modèle de parallélisme à mémoire partagée (style impératif) ou du modèle à mémoire distincte (style purement fonctionnel). La bibliothèque Thread comprend plusieurs modules permettant le lancement et l'arrêt de threads, la gestion de verrou pour l'exclusion mutuelle, l'attente sur condition et la communication entre threads via des canaux propres aux threads. Dans ce modèle, il n'y a pas gain de temps d'exécution d'un programme, y compris sur les machines multi-processeurs. Néanmoins l'expression d'algorithmes parallèles y est grandement facilitée.

Le chapitre 20 se consacre à la fabrication d'applications réparties sur le réseau Internet. Il présente ce réseau du point de vue des protocoles de bas niveau. Grâce aux prises de communication, plusieurs processus s'exécutant sur différentes machines peuvent communiquer entre eux. La communication à travers des sockets est une communication point-à-point asynchrone. Le rôle des différents processus entrant en jeu dans la communication d'une application répartie est en générale asymétrique. C'est le cas dans une architecture client-serveur. Le serveur est un processus acceptant des requêtes et tâchant d'y répondre. Le client, autre processus, envoie une requête au serveur en espérant une réponse. De nombreux services accessibles sur le réseau Internet suivent cette architecture.

Le chapitre 21 présente une bibliothèque et deux applications complètes. La bibliothèque permet de définir la communication entre clients et serveurs à partir d'un protocole donné. La première application reprend les robots du chapitre 17 pour en donner une version répartie. La deuxième application construit un serveur HTTP pour la gestion d'un formulaire de requêtes reprenant les éléments de la gestion d'associations présentée au chapitre 6.

Communication et processus

Introduction

Ce chapitre aborde deux grands traits de l'interfaçage d'un langage de programmation avec le système d'exploitation : la communication et les processus. Le module Sys, présenté au chapitre 8, a déjà montré comment faire passer des valeurs à un programme et comment lancer un programme à partir d'un autre. Le but de ce chapitre est de préciser les notions de processus et de communication entre processus.

Le terme « processus » est utilisé pour un programme en cours d'exécution. Les processus sont les composants premiers des applications parallèles. Nous introduisons les processus sous la présentation classique que l'on donne pour le système Unix. Dans ce cadre, un processus est lancé par un autre processus, créant ainsi un lien de filiation. Il existe un processus premier, ancêtre de tous les autres. Cette relation de paternité autorise des attentes sur la fin de l'exécution d'un processus, ainsi qu'une communication privilégiée entre ascendant et descendant. Le modèle de parallélisme sous-jacent est celui à mémoire répartie.

Le terme « communication » recouvre trois réalités :

- les entrées-sorties au moyen de descripteurs de fichier. La notion de fichier sous Unix a une portée beaucoup plus large que la simple lecture ou écriture de données sur un support de masse. On le verra en particulier au chapitre 20 où des programmes tournant sur diverses machines communiquent via de tels descripteurs ;
- la création et l'utilisation de tubes de communication entre processus qui permettent un échange de données selon le principe des files d'attente;
- la génération et le traitement de signaux qui autorisent une interaction simple entre processus.

Les fonctions présentées dans ce chapitre viennent, pour l'essentiel,

du module Unix accompagnant la distribution d'Objective CAML. La terminologie et les notions utilisées viennent du monde Unix, mais une grande partie des fonctions de ce module sont utilisables sous Windows. Nous préciserons, quand il y aura lieu, le champ d'application des fonctions présentées.

Plan du chapitre

La première section indique comment utiliser le module Unix. Nous y parlons principalement de la gestion des erreurs spécifiques à ce module et de la portabilité sur Windows des appels système.

La deuxième section présente les descripteurs de fichier au sens Unix et leur utilisation pour des opérations d'entrées-sorties de plus bas niveau que celles fournies par le module préchargé Pervasives.

Les processus sont introduits à la troisième section. On y parle de leur création, de leur disparition et de la manière dont tous les processus entretiennent un lien de filiation dans le modèle Unix.

La quatrième section aborde les moyens de base de communication entre processus : les tubes de communication et les signaux.

Ces deux dernières sections se prolongent dans les chapitres [19](#) et [20](#) par la présentation des processus légers et des prises de communication.

Module

On trouve dans ce module l'interface avec les principales fonctions des bibliothèques du système Unix. Cependant, comme nous l'avons déjà signalé, une partie importante de ce module a été portée sous Windows et est donc accessible depuis ce dernier. Nous signalerons, lorsque nécessaire, les restrictions d'utilisation des fonctions présentées. Un tableau résumant ces restrictions est donné figure [18.1](#).

La bibliothèque Unix fait partie des bibliothèques non standard d'Objective CAML qu'il est nécessaire de lier par la commande -custom du compilateur (voir chapitre [7](#), page [??](#)). Suivant l'usage

que l'on destine au programme, on écrit sous Unix l'une des commandes suivantes, pour obtenir du code-octet, du code natif ou une boucle d'interaction :

```
$ ocamlc -custom unix.cma fichiers.ml -cclib -lunix
$ ocamlopt unix.cma fichiers.ml -cclib -lunix
$ ocamlmktop -custom -o unixtop unix.cma -cclib -lunix
```

L'intérêt de construire une boucle d'interaction (dont le nom sera ici `unixtop`) est de permettre un développement incrémentiel. On obtient rapidement la compilation de chaque fonction avec son indication de type. On peut même procéder à quelques tests fonctionnels.

Suivant l'Unix utilisé, la bibliothèque système peut ne pas être à l'emplacement prévu par défaut. Si nécessaire, on peut préciser le chemin d'accès des bibliothèques avec l'option `-ccopt` (voir chapitre 7).

Sous Windows, les commandes de compilation deviennent :

```
$ ocamlc -custom unix.cma fichiers.ml %CAMLLIB%\libunix
$ ocamlopt unix.cma fichiers.ml %CAMLLIB%\libunix.lib ws
$ ocamlmktop -custom -o unixtop.exe unix.cma %CAMLLIB%\l
```

Le nom de la boucle d'interaction ainsi obtenue est `unixtop.exe`.

Gestion des erreurs

Les erreurs provoquées par les appels système déclenchent l'exception `Unix_error` qui peut être traitée depuis un programme Objective CAML. Cette erreur contient trois arguments : une valeur de type `Unix.error` qu'il est possible de traduire sous forme d'une chaîne de caractères par la fonction `error_message`, une chaîne contenant le nom de la fonction ayant provoqué l'erreur et une chaîne contenant éventuellement l'argument de la fonction lorsqu'il est de type *string*.

On peut ainsi définir une fonction générique d'appel de fonction et de traitement des erreurs :

```
# let capsule_unix fonct arg =
  try (fonct arg) with
  Unix.Unix_error (e, fm, argm) ->
  Printf.printf "%s %s %s" (Unix.error_message e) fm argm
val capsule_unix : ('a -> unit) -> 'a -> unit = <fun>
```

La fonction `capsule_unix` prend une fonction, son argument et applique l'une à l'autre. Si une erreur Unix survient, un message explicatif est affiché. Une fonction équivalente est définie dans le module `Unix` :

```
# Unix.handle_unix_error ;;
- : ('a -> 'b) -> 'a -> 'b = <fun>
```

Portabilité des appels système

La figure 18.1 indique quelles fonctions, de communication et de gestion des processus, présentées dans ce chapitre sont accessibles sous Windows. Les deux manques principaux sont la fonction `fork` pour la duplication du processus courant et la fonction `kill` d'émission de signaux.

Fonction	Unix	Windows	Commentaire
<code>openfile</code>	×	×	
<code>close</code>	×	×	
<code>dup</code>	×	×	
<code>dup2</code>	×	×	
<code>read</code>	×	×	
<code>write</code>	×	×	
<code>lseek</code>	×	×	
<code>execv</code>	×	×	
<code>execve</code>	×	×	
<code>execvp</code>	×	×	
<code>execvpe</code>	×	×	
<code>fork</code>	×		utiliser <code>create_process</code>

getpid	×	×	uniquement pour un numéro de processus donné
sleep	×	×	
wait	×		
waitpid	×	×	
create_process	×	×	utilise l'interprète de commandes /bin/sh
create_process_env	×	×	
kill	×		
pipe	×	×	
mkfifo	×		
open_process	×		
close_process	×		

Figure 18.1 : Portabilité des fonctions du module Unix utilisées dans ce chapitre

De plus la fonction wait d'attente de fin d'un processus fils n'est pas implantée car fork ne l'est pas.

Descripteurs de fichier

Au chapitre 3, nous avons vu les fonctions standard du module Pervasives, elles permettent d'accéder aux fichiers via des canaux d'entrées-sorties. Il existe un moyen de plus bas niveau d'accès aux fichiers en ayant recours à leur descripteur.

Un descripteur de fichier est une valeur abstraite, de type `Unix.file_descr`, qui contient les informations nécessaires à l'utilisation d'un fichier : un pointeur vers ce fichier, les droits d'accès à ce fichier, les conditions d'accès au fichier (lecture ou

écriture), la position courante dans le fichier, etc. Trois descripteurs sont prédéfinis. Ils correspondent aux fichiers d'entrée standard, de sortie standard et d'erreur standard.

```
# ( Unix.stdin , Unix.stdout , Unix.stderr ) ;;  
- : Unix.file_descr * Unix.file_descr * Unix.file_descr  
<abstr>, <abstr>, <abstr>
```

Attention à ne pas les confondre avec les canaux d'entrées-sorties leur correspondant :

```
# ( Pervasives.stdin , Pervasives.stdout , Pervasives.stderr ) ;;  
- : in_channel * out_channel * out_channel = <abstr>, <abstr>, <abstr>
```

Des fonctions de conversion entre canaux et descripteurs de fichier sont décrites à la page [??](#).

Droits d'accès aux fichiers

Sous Unix, des droits en lecture, en écriture et en exécution sont attachés à chaque fichier. Ils concernent trois catégories d'utilisateurs : le propriétaire du fichier, les membres du groupe¹ du propriétaire ou l'ensemble de tous les utilisateurs.

Les droits d'un fichier sont représentés par 9 bits divisés en trois groupes de trois bits. Le premier groupe représente les droits du propriétaire, le deuxième les droits des membres du groupe du propriétaire et le dernier les droits des autres utilisateurs. Dans chaque groupe de trois bits, le premier représente le droit en lecture, le deuxième, le droit en écriture et le dernier, le droit en exécution. On a coutume de représenter ces droits respectifs par les lettres *r*, *w* et *x*, et l'absence de droit par un tiret (-). Par exemple, le droit en lecture pour tous et en écriture pour le propriétaire seul s'écrit *rw-r--r--*. C'est, en fait, l'entier *420* (soit le binaire *0b110100100*). On utilisera souvent la notation octale *0o644* qui est d'un emploi plus aisé. Ces droits sur les fichiers ne sont pas utilisés sous Windows.

Manipulation des fichiers

Ouverture d'un fichier

Ouvrir un fichier, c'est récupérer un descripteur de ce fichier. Suivant l'usage que l'on veut en faire, il existe plusieurs modes d'ouverture des fichiers. Chacun correspond à une valeur du type *open_flag* décrit figure 18.2.

O_RDONLY	en lecture
O_WRONLY	en écriture
O_RDWR	en lecture et en écriture
O_NONBLOCK	ouverture dans un mode non bloquant
O_APPEND	en ajout à la fin du fichier
O_CREAT	crée le fichier s'il n'existe pas
O_TRUNC	remet le fichier à 0 s'il existe
O_EXCL	échoue si le fichier existe

Figure 18.2 : Valeurs de type *open_flag*

Ces modes peuvent être combinés, en conséquence la fonction *openfile* prendra en argument une liste de valeurs de type *open_flag*.

```
# Unix.openfile ;;  
- : string -> Unix.open_flag list -> Unix.file_perm -> U  
<fun>
```

Le premier argument est le nom du fichier et le dernier est un entier² codant les permissions à donner au fichier en cas de création.

Voici comment ouvrir en lecture un fichier, ou le créer avec les droits `rw-r--r--` s'il n'existe pas :

```
# let fichier = Unix.openfile "essai.dat" [Unix.O_RDWR;  
val fichier : Unix.file_descr = <abstr>
```

Fermeture d'un fichier

La fonction `Unix.close` permet de fermer un fichier. On l'applique au descripteur du fichier que l'on désire fermer.

```
# Unix.close ;;  
- : Unix.file_descr -> unit = <fun>  
# Unix.close fichier ;;  
- : unit = ()
```

Redirection de fichiers

On peut attacher à une même entrée-sortie plusieurs descripteurs. Si l'on dispose d'un seul descripteur et que l'on veut en obtenir un nouveau, on utilisera :

```
# Unix.dup ;;  
- : Unix.file_descr -> Unix.file_descr = <fun>
```

Si l'on dispose de deux descripteurs et que l'on veut réassigner au second l'entrée-sortie correspondant au premier, on utilisera la fonction :

```
# Unix.dup2 ;;  
- : Unix.file_descr -> Unix.file_descr -> unit = <fun>
```

Par exemple, on redirige la sortie d'erreur sur un fichier :

```
# let sortie_erreur = Unix.openfile "err.log" [Unix.O_WRONLY;  
val sortie_erreur : Unix.file_descr = <abstr>  
# Unix.dup2 Unix.stderr sortie_erreur ;;  
- : unit = ()
```


Les écritures sur la sortie erreur standard sont détournées vers le fichier err.log.

Entrées-sorties sur un fichier

Les fonctions de lecture et d'écriture sur un fichier Unix.read et Unix.write utilisent une chaîne de caractères comme média entre le fichier et le programme Objective CAML.

```
# Unix.read ;;
- : Unix.file_descr -> string -> int -> int -> int = <fu
# Unix.write ;;
- : Unix.file_descr -> string -> int -> int -> int = <fu
```

Outre le descripteur de fichier et la chaîne, les fonctions prennent deux entiers en argument qui sont l'indice du premier caractère et le nombre souhaité de caractères à lire ou à écrire. L'entier retourné est le nombre de caractères effectivement lus ou écrits.

```
# let mode = [Unix.O_WRONLY;Unix.O_CREAT;Unix.O_TRUNC] i
  let fic = Unix.openfile "fichier" mode 0o644 in
  let str = "012345678901234565789" in
  let n = Unix.write fic str 4 5
  in Printf.printf "On a écrit %s dans le fichier\n" (Str
  Unix.close fic ;;
On a écrit 45678 dans le fichier
- : unit = ()
```

La lecture procède du même principe :

```
# let fic = Unix.openfile "fichier" [Unix.O_RDONLY] 0o64
  let str = String.make 20 '.' in
  let n = Unix.read fic str 2 10 in
  Printf.printf "On n'a lu que %d caractères" n;
  Printf.printf " et on obtient la chaîne %s\n" str;
  Unix.close fic ;;
On n'a lu que 5 caractères et on obtient la chaîne ..456
- : unit = ()
```

Un accès à un fichier se fait toujours à la position courante

contenue dans son descripteur. On peut cependant modifier cette position grâce à la fonction :

```
# Unix.lseek ;;  
- : Unix.file_descr -> int -> Unix.seek_command -> int =
```

Le premier argument est le descripteur du fichier, le second est la taille, en caractères, du déplacement et le troisième argument, de type *Unix.seek_command*, indique l'origine. Ce dernier argument a trois valeurs possibles :

- `SEEK_SET` : relativement au début du fichier,
- `SEEK_CUR` : relativement à la position courante,
- `SEEK_END` : relativement à la fin du fichier.

Une position erronée provoque soit le déclenchement d'une exception, soit une valeur de retour égale à 0 suivant les fonctions d'entrées-sorties utilisées.

Canal d'entrée-sortie

Le module Unix fournit des fonctions de conversion entre les descripteurs de fichier et les canaux d'entrées-sorties du module Pervasives :

```
# Unix.in_channel_of_descr ;;  
- : Unix.file_descr -> in_channel = <fun>  
# Unix.out_channel_of_descr ;;  
- : Unix.file_descr -> out_channel = <fun>  
# Unix.descr_of_in_channel ;;  
- : in_channel -> Unix.file_descr = <fun>  
# Unix.descr_of_out_channel ;;  
- : out_channel -> Unix.file_descr = <fun>
```

Il est nécessaire de préciser si les canaux d'entrées-sorties obtenus par conversion transfèrent des données binaires ou des caractères :

```
# set_binary_mode_in ;;  
- : in_channel -> bool -> unit = <fun>  
# set_binary_mode_out ;;
```

```
- : out_channel -> bool -> unit = <fun>
```

Dans l'exemple suivant on crée un fichier en utilisant les fonctions du module Unix et on le lit en utilisant la fonction d'ouverture du module Unix et la fonction d'entrée de plus haut niveau input_line.

```
# let mode = [Unix.O_WRONLY;Unix.O_CREAT;Unix.O_TRUNC] in
  let f = Unix.openfile "fichier" mode 0o666 in
  let s = "0123456789\n0123456789\n" in
  let n = Unix.write f s 0 (String.length s)
  in Unix.close f ;;
- : unit = ()
# let f = Unix.openfile "fichier" [Unix.O_RDONLY;Unix.O_
  let c = Unix.in_channel_of_descr f in
  let s = input_line c
  in print_string s ;
  close_in c ;;
0123456789- : unit = ()
```

Disponibilité

Un programme peut avoir à gérer une multiplicité d'entrées-sorties. Or celles-ci ne sont pas toujours disponibles : d'autres programmes ont pu précédemment en réclamer l'usage. La fonction suivante permet de connaître la liste des entrées-sorties disponibles à un moment donné parmi une liste donnée :

```
# Unix.select ;;
- : Unix.file_descr list ->
  Unix.file_descr list ->
  Unix.file_descr list ->
  float ->
  Unix.file_descr list * Unix.file_descr list * Unix.file
= <fun>
```

Les trois premiers arguments représentent les listes respectivement des entrées (lecture), des sorties (écriture) et des sorties en erreur. Le dernier argument indique un délai d'attente en secondes (une

valeur négative indique un délai nul). Le résultat est la liste des entrées-sorties disponibles en lecture, écriture et erreur.

Warning

select n'est pas implantée pour Windows

Processus

Unix est un système qui associe à chaque exécution d'un programme un *processus*. Dans [CDM96] Card, Dumas et Mével résument ainsi la différence entre programme et processus : < < un programme en lui-même n'est pas un processus : un programme est une entité passive (un fichier exécutable résidant sur un disque), alors qu'un processus est une entité active avec un compteur ordinal spécifiant l'instruction suivante à exécuter et un ensemble de ressources associées. > >

Unix est un système dit multi-tâches : plusieurs processus peuvent être exécutés simultanément. Il est préemptif, l'exécution des processus est confiée à un processus particulier chargé de leur ordonnancement. Un processus n'est donc pas entièrement maître de ses ressources. Au premier chef, un processus n'est pas maître du moment de son exécution et ce n'est pas parce qu'il est créé qu'un processus est exécuté sur le champ.

Chaque processus dispose de son propre espace mémoire. Les processus peuvent communiquer à travers des fichiers ou des tubes de communication. Nous sommes en présence du modèle de parallélisme à mémoire distribuée simulé sur une seule machine.

Le système attribue aux processus un identificateur unique : un entier appelé PID (Process IDentifier). De plus, sous Unix, à l'exception notable du processus initial, tout processus est engendré par un autre processus que l'on appelle son père.

On peut connaître l'ensemble des processus actifs par la commande Unix `ps`³ :

```
$ ps -f
PID      PPID      CMD
1767      1763      csh
2797      1767      ps -f
```

L'emploi de l'option -f fait apparaître, pour chaque processus actif, son identificateur (PID), celui de son père (PPID) et le programme invoqué (CMD). Ici, nous avons deux processus, l'interprète de commandes csh et la commande ps elle-même. On note que ps étant invoquée depuis l'interprète de commandes csh, le père de son processus est le processus associé à l'exécution de csh.

Exécution d'un programme

Environnement d'exécution

Trois valeurs sont associées à un programme exécuté depuis un interprète de commandes du système d'exploitation :

1. la ligne de commande ayant servi à son exécution qui est contenue dans la valeur Sys.argv,
2. les variables d'environnement de l'interprète de commandes que l'on peut récupérer grâce à la fonction Sys.getenv,
3. un statut d'exécution lorsque le programme termine.

Ligne de commande

La ligne de commande permet de récupérer les arguments ou options d'appel d'un programme. Celui-ci peut alors déterminer son comportement en fonction de ces valeurs. En voici un petit exemple. On écrit le petit programme suivant dans le fichier argv_ex.ml :

```
if Array.length Sys.argv = 1 then
  Printf.printf "Hello world\n"
else if Array.length Sys.argv = 2 then
```

```
Printf.printf "Hello %s\n" Sys.argv.(1)
else Printf.printf "%s : trop d'arguments\n" Sys.argv.(0)
```

On le compile :

```
$ ocamlc -o argv_ex argv_ex.ml
```

Et on exécute successivement :

```
$ argv_ex
Hello world
$ argv_ex lecteur
Hello lecteur
$ argv_ex cher lecteur
./argv_ex : trop d'arguments
```

Variables d'environnement

Les variables d'environnement contiennent différentes valeurs nécessaires à la bonne marche du système ou de certaines applications. Le nombre et le nom de ces variables dépendent à la fois du système d'exploitation et de configurations propres aux utilisateurs. Le contenu de ces variables est accessible par la fonction `getenv` qui prend en argument le nom d'une variable d'environnement sous forme d'une chaîne de caractères :

```
# Sys.getenv "HOSTNAME";;
- : string = "zinc.pps.jussieu.fr"
```

Statut d'exécution

La valeur de retour d'un programme est un entier fixé, en général, automatiquement par le système suivant que le programme se termine en erreur ou non. Le développeur peut toujours mettre fin explicitement à son programme en précisant la valeur du statut d'exécution par appel à la fonction :

```
# Pervasives.exit ;;
- : int -> 'a = <fun>
```

Lancement de processus

Un programme est lancé à partir d'un processus que l'on appelle le processus courant. L'exécution du programme devient un nouveau processus. On obtient trois cas de figure :

- les deux processus sont indépendants et peuvent s'exécuter concurremment;
- le processus père est en attente sur la fin d'exécution du processus fils;
- le processus lancé remplace le processus père qui disparaît.

On peut aussi dupliquer le processus courant et obtenir ainsi deux instances du même processus qui ne diffèrent que par leur PID. C'est le fameux fork que nous décrivons dans la suite.

Processus indépendants

Le module Unix offre une fonction portable de lancement d'un processus correspondant à l'exécution d'un programme.

```
# Unix.create_process ;;  
- : string ->  
  string array ->  
  Unix.file_descr -> Unix.file_descr -> Unix.file_descr -  
= <fun>
```

Le premier argument est le nom du programme (qui peut être un chemin), le deuxième, le tableau d'arguments du programme, les trois derniers sont les descripteurs devant servir à l'entrée standard, à la sortie standard et à la sortie en erreur du processus. La valeur de retour est le numéro du processus créé.

Il existe une variante de cette fonction permettant de préciser la valeur de variables d'environnement :

```
# Unix.create_process_env ;;
- : string ->
  string array ->
  string array ->
  Unix.file_descr -> Unix.file_descr -> Unix.file_descr -
= <fun>
```

Ces deux fonctions sont utilisables sous Unix ou Windows.

Empilement de processus

Il n'est pas toujours utile que le processus lancé le soit de manière concurrente. En effet le processus père peut avoir besoin d'attendre la fin du processus qu'il vient de lancer pour poursuivre sa tâche. Les deux fonctions suivantes prennent comme argument le nom d'une commande et l'exécutent.

```
# Sys.command;;
- : string -> int = <fun>
# Unix.system;;
- : string -> Unix.process_status = <fun>
```

Elles diffèrent par le type du code retour. Le type *process_status* est détaillé à la page [??](#). Pendant l'exécution de la commande le processus père est bloqué.

Remplacement de processus courant

Le remplacement du processus courant par la commande qu'il vient de lancer permet de limiter le nombre de processus en cours d'exécution. Les quatre fonctions suivantes effectuent ce travail :

```
# Unix.execv ;;
- : string -> string array -> unit = <fun>
# Unix.execve ;;
- : string -> string array -> string array -> unit = <fu
# Unix.execvp ;;
```



```
- : string -> string array -> unit = <fun>
# Unix.execvpe ;;
- : string -> string array -> string array -> unit = <fu
```

Leur premier argument est le nom du programme. En utilisant `execvp` ou `execvpe`, ce nom peut indiquer un chemin dans l'arborescence des fichiers. Le second argument contient les arguments du programme qu'il est possible de passer sur la ligne de commande. Le dernier argument des fonctions `execve` et `execvpe` permet en plus d'indiquer la valeur des variables système utiles au programme.

Création d'un processus par duplication

L'appel système originel de création de processus sous Unix est :

```
# Unix.fork ;;
- : unit -> int = <fun>
```

La fonction `fork` engendre un nouveau processus et non un nouveau programme. Son effet exact est de *dupliquer* le processus appelant. Le code du nouveau processus est le même que celui de son père. Sous Unix un même code peut servir à plusieurs processus, chacun possédant son propre contexte d'exécution. On parle alors de code réentrant.

Voyons cela sur le petit programme suivant (on utilise la fonction `getpid` qui retourne le PID du processus associé à l'exécution du programme) :

```
Printf.printf "avant fork : %d\n" (Unix.getpid ()) ;;
flush stdout ;;
Unix.fork () ;;
Printf.printf "après fork : %d\n" (Unix.getpid ()) ;;
flush stdout ;;
```

On obtient l'affichage suivant :

```
avant fork : 1447
```

```
après fork : 1447
```

```
après fork : 1448
```

Après l'exécution du fork, deux processus exécutent la suite du code. C'est ce qui provoque deux fois l'affichage du PID « après ». On remarque qu'un des processus a gardé le PID de départ (le père) alors que l'autre en a un nouveau (le fils) qui correspond à la valeur de retour de l'appel à fork. Pour le processus père la valeur de retour de fork est le PID du fils alors que pour le fils, elle vaut 0.

C'est cette différence de valeur de retour de fork qui permet, *dans un même programme source*, de différencier le code exécuté par le fils du code exécuté par le père :

```
Printf.printf "avant fork : %d\n" (Unix.getpid ()) ;;
flush stdout ;;
let pid = Unix.fork () ;;
if pid=0 then (* -- Code du fils *)
  Printf.printf "je suis le fils : %d\n" (Unix.getpid ())
else (* -- Code du pere *)
  Printf.printf "je suis le père : %d du fils : %d\n" (U
flush stdout ;;
```

Voici la trace de l'exécution de ce programme :

```
avant fork : 1456
```

```
je suis le père : 1456 du fils : 1457
```

```
je suis le fils : 1457
```

On peut aussi utiliser la valeur de retour dans un filtrage :

```
match Unix.fork () with
0 -> Printf.printf "je suis le fils : %d\n" (Unix.getpid ())
| pid -> Printf.printf "je suis le père : %d du fils : %d\n"
(Unix.getpid ()) pid ;;
```

La fécondité d'un processus peut être très grande. Elle est cependant limitée à un nombre fini de descendants par la configuration du système d'exploitation. L'exemple suivant crée deux générations de

processus avec grand père, pères, fils, oncles et cousins.

```
let pid0 = Unix.getpid ();;  
let print_generation1 pid ppid =  
  Printf.printf "Je suis %d, fils de %d\n" pid ppid;  
  flush stdout ;;  
  
let print_generation2 pid ppid pppid =  
  Printf.printf "Je suis %d, fils de %d, petit fils de %d\n" pid ppid pppid;  
  flush stdout ;;  
  
match Unix.fork() with  
  0 -> let pid01 = Unix.getpid ()  
    in ( match Unix.fork() with  
      0 -> print_generation2 (Unix.getpid ()) pid01 pid0  
      | _ -> print_generation1 pid01 pid0)  
      | _ -> match Unix.fork() with  
        0 -> ( let pid02 = Unix.getpid ()  
          in match Unix.fork() with  
            0 -> print_generation2 (Unix.getpid ()) pid02 pid0  
            | _ -> print_generation1 pid02 pid0 )  
            | _ -> Printf.printf "Je suis %d, père et grand père\n"
```

On obtient :

```
Je suis 1548, père et grand père  
Je suis 1549, fils de 1548  
Je suis 1550, fils de 1548  
Je suis 1552, fils de 1549, petit fils de 1548  
Je suis 1553, fils de 1550, petit fils de 1548
```

Ordre et moment d'exécution

En enchaînant sans précaution la création de processus, on peut obtenir des effets poétiques à la M. Jourdain :

```
match Unix.fork () with
  0 -> Printf.printf "Marquise " ; flush stdout
| _ -> match Unix.fork () with
  0 -> Printf.printf "vos beaux yeux me font " ; flush st
| _ -> Printf.printf "mourir d'amour\n" ; flush stdout
```

Ce qui peut donner le résultat suivant :

```
mourir d'amour
Marquise vos beaux yeux me font
```

Pour obtenir un M. Jourdain prosateur, il faut que notre programme soit capable de s'assurer lui-même de l'ordre d'exécution des processus le composant. De façon plus générale, lorsqu'une application met en oeuvre plusieurs processus elle doit, quand besoin est, s'assurer de leur synchronisation. Selon le modèle de parallélisme utilisé, cette synchronisation est réalisée par la communication entre processus ou par des attentes sur condition. Cette problématique est plus amplement présentée dans les deux prochains chapitres. En attendant, on peut rendre M. Jourdain prosateur de deux façons :

- laisser du temps au fils pour écrire son bout de phrase avant d'écrire le sien propre.
- attendre la mort du fils qui aura écrit son bout de phrase avant d'écrire son propre bout de phrase.

Délai d'attente

Un processus peut suspendre son activité en appelant la fonction :

```
# Unix.sleep ;;
- : int -> unit = <fun>
```

L'argument fournit le nombre de secondes de suspension du processus appelant avant la reprise d'activité.

En utilisant cette fonction, on écrira :

```
match Unix.fork () with
  0 -> Printf.printf "Marquise " ; flush stdout
  | _ -> Unix.sleep 1 ;
match Unix.fork () with
  0 -> Printf.printf"vos beaux yeux me font " ; flush stdout
  | _ -> Unix.sleep 1 ; Printf.printf"mourir d'amour\n" ;
```

Et on pourra obtenir :

```
Marquise vos beaux yeux me font mourir d'amour
```

Néanmoins, cette méthode n'est pas sûre. A priori, rien n'empêche le système d'allouer suffisamment de temps à l'un des processus pour qu'il puisse à la fois réaliser son temps de sommeil et son affichage. Nous préférons donc dans notre cas la méthode ci-dessous qui séquentialise les processus.

Attente de terminaison du fils

Un processus père a la possibilité d'attendre la mort de son fils par appel à la fonction :

```
# Unix.wait ;;
- : unit -> int * Unix.process_status = <fun>
```

L'exécution du père est suspendue jusqu'à terminaison de l'un de ses fils. Si un wait est exécuté par un processus n'ayant plus de fils, alors l'exception `Unix_error` est déclenchée. Nous reviendrons ultérieurement sur la valeur de retour de `wait`. Ignorons la pour l'instant et faisons dire de la prose à M. Jourdain :

```
match Unix.fork () with
```

```
0 -> Printf.printf "Marquise " ; flush stdout
| _ -> ignore (Unix.wait ()) ;
match Unix.fork () with
0 -> Printf.printf "vos beaux yeux me font " ; flush st
| _ -> ignore (Unix.wait ()) ;
Printf.printf "mourir d'amour\n" ;
flush stdout
```

Et, de fait, il dit :

Marquise vos beaux yeux me font mourir d'amour

Warning

fork est propre au système Unix

Filiation, mort et funérailles d'un processus

La fonction wait n'a pas pour seule utilité l'attente de terminaison du fils utilisée ci-dessus. Elle est également chargée de consacrer la mort du processus fils.

Lorsqu'un processus est créé, le système ajoute une entrée dans la table qui lui sert à gérer l'ensemble des processus. Lorsqu'il meurt, un processus ne disparaît pas automatiquement de cette table. C'est au père, par appel à wait, d'assurer la suppression de ses fils de la table des processus. S'il ne le fait pas, le processus fils subsiste dans la table des processus. On parle alors de processus zombie. Au démarrage du système, est lancé un premier processus appelé init. Après initialisation d'un certain nombre de paramètres du système, un rôle essentiel de ce « grand ancêtre » est de prendre en charge les processus orphelins et d'exécuter le wait qui les rayera de la table des processus à leur terminaison.

Attente de fin d'un processus donné

Il existe une variante de la fonction `wait`, nommée `waitpid` et portée sous Windows :

```
# Unix.waitpid ;;  
- : Unix.wait_flag list -> int -> int * Unix.process_status
```

Le premier argument spécifie les modalités d'attente et le second, quel processus ou quel groupe de processus il faut traiter.

À la mort d'un processus, deux informations sont accessibles au père comme résultat de l'appel à `wait` ou `waitpid` : le numéro du processus terminé et son statut de terminaison. Ce dernier est représenté par une valeur de type *Unix.process_status*. Ce type a trois constructeurs dont chacun a un argument entier.

- *WEXITED* n : le processus s'est terminé normalement avec le code de retour n .
- *WSIGNALED* n : le processus a été tué par le signal n .
- *WSTOPPED* n : le processus a été stoppé par le signal n .

La dernière valeur n'a de sens que pour la fonction `waitpid` qui peut, par son premier argument, se mettre à l'écoute de tels signaux. Nous détaillons les signaux et leur traitement page ??.

Gestion de l'attente par un ancêtre

Pour éviter de gérer soi-même la terminaison d'un processus fils, il est possible de faire traiter cette attente par un processus ancêtre. C'est l'astuce du « double fork » qui permet à un processus de n'avoir pas à se soucier des funérailles de ses descendants en les confiant au processus init. En voici le principe : un processus P_0 crée un processus P_1 qui à son tour crée un troisième processus P_2 puis termine. Ainsi, P_2 se retrouve orphelin et est adopté par init qui se chargera d'attendre sa terminaison. Le processus initial P_0 peut alors exécuter un `wait` sur P_1 qui sera de courte durée. L'idée est de confier au petit fils le travail que l'on aurait confié au fils.

Le schéma de mise en oeuvre est le suivant :

```
# match Unix.fork() with (* P0 crée P1 *)
  0 -> if Unix.fork() = 0 then exit 0 ;                               (* P1 cr
  Printf.printf "P2 fait son travail\n" ;
  exit 0
  | pid -> ignore (Unix.waitpid [] pid) ;                               (* P0
  Printf.printf "P0 peut faire autre chose sans attendre\
P2 fait son travail
P0 peut faire autre chose sans attendre
- : unit = ()
```

Nous verrons une utilisation de ce principe pour traiter les requêtes envoyées à un serveur au chapitre [20](#).

Communication entre processus

L'utilisation de processus dans le développement d'une application permet la délégation des tâches. Néanmoins, comme nous l'avons déjà évoqué, ces tâches peuvent ne pas être indépendantes et il est dès lors nécessaire que les processus sachent communiquer entre eux.

Nous abordons dans ce paragraphe deux modes de communication entre processus : les tubes de communication (pipes) et les signaux. Ce chapitre ne fait pas un tour complet des possibilités de communication entre processus. Il n'est qu'une première approche des applications développées aux chapitres [19](#) et [20](#).

Tubes de communication

À la manière des fichiers, il est possible de communiquer directement entre processus à travers des tubes de communication.

Les tubes sont en quelque sorte des fichiers virtuels dans lesquels on peut lire et écrire au moyen des fonctions d'entrées-sorties `read` et `write`. Cependant ils sont de taille limitée (cette limite dépendant des systèmes) et leur discipline de remplissage et de vidage est celle

des files d'attente : le premier entré est le premier sorti. Et quand nous disons « sorti », il faut prendre l'expression au pied de la lettre : la lecture de données dans un tube les supprime de celui-ci.

Cette discipline de file d'attente est réalisée en associant deux descripteurs à un tube : l'un correspond à l'extrémité du tube dans laquelle on écrit ; l'autre, l'extrémité dans laquelle on lit. Un tube est créé par la fonction :

```
# Unix.pipe ;;  
- : unit -> Unix.file_descr * Unix.file_descr = <fun>
```

La première composante du couple obtenu en résultat est la sortie du tube utilisée en lecture et la seconde, l'entrée du tube utilisée en écriture. Tout processus en ayant connaissance peut fermer ces descripteurs.

La lecture dans un tube est bloquante sauf si tous les processus connaissant son descripteur d'entrée (et donc, susceptible d'y écrire) l'ont fermé. Dans ce dernier cas, la fonction `read` renvoie 0. Si un processus tente d'écrire dans un tube plein, il est suspendu jusqu'à ce qu'un autre processus ait effectué une lecture. Si un processus tente d'écrire dans un tube alors que plus aucun autre processus n'est susceptible d'y lire (tous ont fermé le descripteur de sortie) alors le processus tentant d'écrire reçoit le signal `sigpipe` qui, sauf mention contraire, provoque sa terminaison.

L'exemple suivant montre l'utilisation des tubes dans lesquels des petits fils communiquent leur numéro à leur grand père.

```
let sortie, entree = Unix.pipe();;
```

```
let write_pid entree =
```

```
  try
```

```
    let m = "(" ^ (string_of_int (Unix.getpid ())) ^ ")"
```

```
  in ignore (Unix.write entree m 0 (String.length m)) ;
```

```
  Unix.close entree
```

```
  with
```

```
    Unix.Unix_error(n,f,arg) ->
```

```
    Printf.printf "%s(%s) : %s\n" f arg (Unix.error_message
```

```

match Unix.fork () with
  0 -> for i=0 to 5 do
    match Unix.fork() with
      0 -> write_pid entree ; exit 0
      | _ -> ()
    done ;
  Unix.close entree
  | _ -> Unix.close entree;
let s = ref "" and buff = String.create 5
in while true do
  match Unix.read sortie buff 0 5 with
    0 -> Printf.printf "Mes petits fils sont %s\n" !s ; exit 0
    | n -> s := !s ^ (String.sub buff 0 n) ^ "."
  done ;;

```

On obtient la trace :

```
Mes petits fils sont (1575.).(1576.).(1577.).(1578.).(1579.)
```

Nous avons introduit des points entre chaque partie de chaîne lue. On peut ainsi lire sur la trace la succession des contenus du tube. On remarquera que la lecture peut ainsi être désynchronisée : dès qu'une entrée, fût-elle partielle, est produite, elle est consommée.

Les tubes nommés

Certains Unix acceptent de nommer les tubes comme s'il s'agissait de fichiers normaux. Il est alors possible de communiquer entre deux processus sans lien de parenté en utilisant le nom du tube. La fonction suivante permet de créer le tube.

```

# Unix.mkfifo ;;
- : string -> Unix.file_perm -> unit = <fun>

```

Les descripteurs de fichier pour l'utiliser sont obtenus par `openfile` comme s'il s'agissait de fichiers, mais leur comportement est celui des pipes. En particulier, puisqu'il s'agit de files d'attente, on ne peut appliquer la fonction `lseek` à un tube.

Warning

mkfifo n'est pas implantée pour Windows.

Canaux de communication

Le module Unix fournit une fonction de haut niveau permettant le lancement d'un programme en lui associant des canaux d'entrée ou de sortie avec le programme appelant :

```
# Unix.open_process ;;  
- : string -> in_channel * out_channel = <fun>
```

L'argument est le nom du programme ou, plus précisément, la ligne d'appel du programme telle qu'on la taperait pour l'interprète de commande. Elle pourra donc éventuellement contenir les arguments du programme à lancer. Les deux valeurs de sortie sont les descripteurs de fichier associés aux entrées-sorties standard du programme ainsi lancé qui est exécuté en parallèle du programme appelant.

Warning

Le programme lancé par `open_process` est exécuté par appel à l'interprète de commandes Unix `/bin/sh`.

L'utilisation de cette fonction n'est possible que sur les systèmes connaissant cet interprète de commandes.

On peut mettre fin à l'exécution d'un programme lancé par `open_process` en utilisant :

```
# Unix.close_process ;;  
- : in_channel * out_channel -> Unix.process_status = <f
```

L'argument est le couple des canaux associés à un processus que l'on veut fermer. La valeur de retour est le statut d'exécution du

processus dont on a attendu la terminaison.

Il existe des variantes de ces deux fonctions n'ouvrant et ne fermant qu'un canal d'entrée ou un canal de sortie :

```
# Unix.open_process_in ;;
- : string -> in_channel = <fun>
# Unix.close_process_in ;;
- : in_channel -> Unix.process_status = <fun>
# Unix.open_process_out ;;
- : string -> out_channel = <fun>
# Unix.close_process_out ;;
- : out_channel -> Unix.process_status = <fun>
```

Voici un petit exemple amusant d'utilisation de `open_process` : on lance `ocaml` dans `ocaml` !

```
# let n_print_string s = print_string s ; print_string "\n"
val n_print_string : string -> unit = <fun>
# let p () =
  let oc_in, oc_out = Unix.open_process "/usr/local/bin/ocaml"
  in n_print_string (input_line oc_in) ; print_newline() ;
  n_print_string (input_line oc_in) ; print_newline() ;
  print_char (input_char oc_in) ;
  print_char (input_char oc_in) ;
  flush stdout ;
  let s = input_line stdin
  in output_string oc_out s ;
  output_string oc_out "#quit\059\059\n" ;
  flush oc_out ;
  let r = String.create 250 in
  let n = input oc_in r 0 250
  in n_print_string (String.sub r 0 n) ;
  print_string "Merci de votre visite\n" ;
  flush stdout ;
  Unix.close_process (oc_in, oc_out) ;;
val p : unit -> Unix.process_status = <fun>
```

L'appel de la fonction `p` lance un toplevel d'Objective CAML. On remarquera que c'est la version 2.03 qui se trouve dans le catalogue `/usr/local/bin`. Les quatre premières opérations de lecture

permettent de récupérer l'en-tête qu'affiche le toplevel. La ligne `let x = 1.2 +. 5.6;;` est lue au clavier, puis envoyée sur `oc_out` (le canal de sortie lié à l'entrée standard du nouveau processus). Celui-ci type et évalue la phrase Objective CAML passée, et écrit le résultat dans sa sortie standard qui est liée au canal d'entrée `oc_in`. Ce résultat est alors lu par la fonction `input` et affiché, ainsi que la chaîne "Merci de votre visite". On envoie d'autre part la directive `#quit;;` pour sortir du nouveau processus.

```
# p();;
      Objective Caml version 2.03

# let x = 1.2 +. 5.6;;
val x : float = 6.8
Merci de votre visite
- : Unix.process_status = Unix.WSIGNALED 13
#
```

Signaux sous Unix

Une des possibilités pour communiquer avec un processus est de lui envoyer un signal. Un signal peut être reçu à n'importe quel moment de l'exécution du programme. La réception d'un signal provoque une interruption logicielle. L'exécution d'un programme est interrompue pour traiter le signal reçu, puis reprend à l'endroit de son interruption. Les signaux sont en nombre fini et relativement restreint (32 avec Linux). L'information véhiculée par un signal est rudimentaire : elle se borne à l'identité (le numéro) du signal. Les processus ont tous une réaction prédéfinie aux signaux. Néanmoins, celle-ci peut être redéfinie par le programmeur pour la plupart des signaux.

Les données et fonctions de traitement des signaux sont réparties entre les modules `Sys` et `Unix`. Le module `Sys` contient la déclaration d'un certain nombre de signaux répondant à la norme POSIX (décrits dans [\[Rif90\]](#)) ainsi que des fonctions de traitement des signaux. Le module `Unix` définit la fonction `kill` d'émission d'un signal. L'utilisation des signaux sous Windows est restreinte au seul

sigint.

Un signal peut avoir de multiples sources : le clavier ; une tentative erronée d'accès mémoire, etc. Un processus peut émettre un signal à destination d'un autre processus en ayant recours à la fonction :

```
# Unix.kill ;;  
- : int -> int -> unit = <fun>
```

Son premier paramètre est le PID du processus destinataire et le second est le signal qu'on veut lui envoyer.

Traitement des signaux

La réaction associée à un signal peut être de trois ordres. À chacun d'eux correspond un constructeur du type `signal_behavior` :

- `Signal_default` : le comportement par défaut défini par le système. Dans la plupart des cas c'est la terminaison du processus avec création ou non d'un fichier d'état du processus (fichier core).
- `Signal_ignore` : le signal est ignoré.
- `Signal_handle` : le comportement est redéfini par une fonction Objective CAML de type `int -> unit` que l'on passe en argument au constructeur. Lors du traitement du signal ainsi modifié, le numéro de signal est passé à la fonction de traitement.

À la réception d'un signal, l'exécution du processus récepteur est déroutée vers la fonction de traitement du signal. La fonction permettant de redéfinir le comportement associé à un signal est fournie par le module `Sys` :

```
# Sys.set_signal;;  
- : int -> Sys.signal_behavior -> unit = <fun>
```

Le premier argument est le signal à redéfinir et le second, le comportement assigné.

Le module Sys fournit une autre fonction de modification du traitement des signaux :

```
# Sys.signal ;;  
- : int -> Sys.signal_behavior -> Sys.signal_behavior =
```

Elle agit comme `set_signal`, sauf qu'en plus elle renvoie la valeur associée au signal avant la modification. On peut ainsi écrire une fonction renvoyant (sans la modifier apparemment) la valeur comportementale associée à un signal :

```
# let signal_behavior s =  
  let b = Sys.signal s Sys.Signal_default  
  in Sys.set_signal s b ; b ;;  
val signal_behavior : int -> Sys.signal_behavior = <fun>  
# signal_behavior Sys.sigint;;  
- : Sys.signal_behavior = Sys.Signal_handle <fun>
```

Certains signaux ne peuvent pas voir leur comportement modifié. Notre fonction n'est donc pas utilisable pour n'importe quel signal :

```
# signal_behavior Sys.sigkill ;;  
Uncaught exception: Sys_error("Invalid argument")
```

Quelques signaux

Nous illustrons ci-dessous l'utilisation de quelques signaux essentiels.

sigint

Ce signal est, en général, associé à la combinaison de touches CTRL-C. Dans le petit exemple ci-dessous, nous modifions la réaction à ce signal de façon à ce que le processus récepteur ne s'interrompe qu'à la troisième occurrence du signal.

Créons le fichier `ctrlc.ml` suivant :

```
let sigint_handle =
```

```

let n = ref 0
in function _ -> incr n ;
match !n with
  1 -> print_string "Vous venez d'appuyer sur CTRL-C\n"
  | 2 -> print_string "Vous avez encore appuyé sur CTRL-C\n"
  | 3 -> print_string "Si vous insistez ...\n" ; exit 1
  | _ -> () ;;
Sys.set_signal Sys.sigint (Sys.Signal_handle sigint_handle)
match Unix.fork () with
  0 -> while true do () done
  | pid -> Unix.sleep 1 ; Unix.kill pid Sys.sigint ;
    Unix.sleep 1 ; Unix.kill pid Sys.sigint ;
    Unix.sleep 1 ; Unix.kill pid Sys.sigint ;;

```

Ce programme simule l'appui de la combinaison de touches CTRL-C par l'envoi du signal sigint, on obtient la trace d'exécution suivante :

```

$ ocamlc -i -o ctrlc ctrlc.ml
val sigint_handle : int -> unit
$ ctrlc
Vous venez d'appuyer sur CTRL-C
Vous avez encore appuyé sur CTRL-C
Si vous insistez ...

```

sigalarm

Un autre signal couramment utilisé est sigalarm qui est associé à l'horloge de la machine. Il peut être émit par la fonction :

```

# Unix.alarm ;;
- : int -> int = <fun>

```

L'argument spécifie le nombre de secondes d'attente avant l'émission du signal sigalarm. La valeur de retour est le nombre de secondes restant à courir avant l'émission d'un prochain signal, ou si aucune alarme n'est en cours.

Nous allons utiliser cette fonction et le signal associé pour définir la

fonction `timeout` qui lance l'exécution d'une autre fonction et l'interrompt, si besoin est, au bout d'un temps donné. Plus précisément, la fonction *timeout* prendra en argument une fonction *f*, l'argument *arg* attendu par *f*, la durée (time) du « *timeout* » et la valeur (*default_value*) à rendre si cette dernière est dépassée.

Dans `timeout`, les choses se passent ainsi :

1. On modifie le comportement associé au signal `sigalrm` de façon à déclencher l'exception `Timeout`.
2. On a pris soin, au passage, de mémoriser le comportement original associé à `sigalrm` pour définir une fonction capable de le restaurer.
3. On déclenche l'horloge.
4. On lance le calcul :
 1. Si tout s'est bien passé, on remet `sigalrm` dans son état d'origine et on renvoie la valeur du calcul.
 2. Sinon, on restaure `sigalrm` et si la durée est dépassée, on renvoie la valeur par défaut.

Voici les définitions correspondantes ainsi qu'un petit essai :

```
# exception Timeout ;;
exception Timeout
# let sigalrm_handler = Sys.Signal_handle (fun _ -> raise
val sigalrm_handler : Sys.signal_behavior = Sys.Signal_h
# let timeout f arg time default_value =
  let old_behavior = Sys.signal Sys.sigalrm sigalrm_handl
  let reset_sigalrm () = Sys.set_signal Sys.sigalrm old_b
  in ignore (Unix.alarm time) ;
  try let res = f arg in reset_sigalrm () ; res
  with exc -> reset_sigalrm () ;
  if exc=Timeout then default_value else raise exc ;;
val timeout : ('a -> 'b) -> 'a -> int -> 'b -> 'b = <fun
# let itere n = for i = 1 to n do () done ; n ;;
val itere : int -> int = <fun>
```

```
Printf.printf "1ère exécution : %d\n" (timeout itere 10
Printf.printf "2ème exécution : %d\n" (timeout itere 100
```

```
1ère exécution : 10
2ème exécution : -1
- : unit = ()
```

sigusr1 et sigusr2

Ces deux signaux sont à la disposition du programmeur pour les besoins de ses applications. Ils ne sont pas utilisés par le système d'exploitation.

Dans cet exemple, la réception, par le fils, du signal sigusr1 provoque l'affichage du contenu de la variable *i*.

```
let i = ref 0 ;;
let affiche_i s = Printf.printf "signal reçu (%d) -- i=%d\n" i s;
  flush stdout ;;
Sys.set_signal Sys.sigusr1 (Sys.Signal_handle affiche_i)

match Unix.fork () with
| 0 -> while true do incr i done
| pid -> Unix.sleep 0 ; Unix.kill pid Sys.sigusr1 ;
  Unix.sleep 3 ; Unix.kill pid Sys.sigusr1 ;
  Unix.sleep 1 ; Unix.kill pid Sys.sigkill
```

Voici la trace d'une exécution de ce programme :

```
signal reçu (10) -- i=0
signal reçu (10) -- i=47580467
```

En examinant cette trace, on voit qu'après avoir exécuté une première fois le code associé au signal sigusr1, le processus fils continue à exécuter la boucle et à incrémenter *i*.

sigchld

Ce signal est émis vers son père à la terminaison d'un processus. Nous allons l'utiliser pour rendre un père plus attentif au devenir de ses enfants. Voici comment :

1. On définit une fonction de traitement du signal `sigchld` qui traite tous les enfants morts à la réception de ce signal⁴ et termine le processus père lorsque celui-ci n'a plus d'enfant (exception `Unix_error`). Pour ne pas bloquer le père, si tous ses enfants ne sont pas morts, on utilise `waitpid` plutôt que `wait`.
2. Le programme principal, après avoir redéfini la réaction associée à `sigchld`, boucle pour créer cinq fils. Ceci fait, le père fait autre chose (boucle **`while true`**) jusqu'à la mort de ses fils.

```
let rec sigchld_handle s =
  try let pid, _ = Unix.waitpid [Unix.WNOHANG] 0
  in if pid <> 0
  then ( Printf.printf "%d est mort et enterré au signal\n" pid ;
        flush stdout ;
        sigchld_handle s )
  with Unix.Unix_error(_, "waitpid", _) -> exit 0 ;;

let i = ref 0
in Sys.set_signal Sys.sigchld (Sys.Signal_handle sigchld_handle)
while true do
  match Unix.fork() with
  0 -> let pid = Unix.getpid ()
  in Printf.printf "Création de %d\n" pid ; flush stdout
  Unix.sleep (Random.int (5+ !i)) ;
  Printf.printf "Terminaison de %d\n" pid ; flush stdout
  exit 0
  | _ -> incr i ; if !i = 5 then while true do () done
done ;;
```

On obtient la trace :

Création de 1561

```
Création de 1562
Création de 1563
Création de 1564
Création de 1565
Terminaison de 1565
1565 est mort et enterré au signal 17
Terminaison de 1561
1561 est mort et enterré au signal 17
Terminaison de 1562
1562 est mort et enterré au signal 17
Terminaison de 1563
1563 est mort et enterré au signal 17
Terminaison de 1564
1564 est mort et enterré au signal 17
```

Exercices

Les trois exercices proposés ici manipulent respectivement les descripteurs de fichier, les processus, les tubes de communication, et les signaux. Les deux premiers sont des classiques de la programmation système sous Unix. Il peut être intéressant de comparer le code Objective CAML avec le code C que l'on trouve dans les distributions d'Unix ou de Linux.

Compter les mots : la commande

On veut (re)programmer la commande Unix `wc` qui compte le nombre de lignes, mots ou caractères que contient un fichier texte. Les mots sont séparés par un espace, une tabulation ou un retour chariot. On ne compte pas les caractères séparateurs.

1. Écrire une première version (`wc1`) de cette commande qui ne traite qu'un seul fichier dont le nom est passé en argument sur la ligne de commande.

```
# let seps = " \t" ;;
val seps : string = " \t"
# let nl = ref 0 ;;
```

```

val nl : int ref = {contents=0}
# let nw = ref 0 ;;
val nw : int ref = {contents=0}
# let nc = ref 0 ;;
val nc : int ref = {contents=0}

```

2. Écrire une version plus élaborée ([wc2](#)) qui peut prendre en argument tout ou partie des trois options : -l, -c, -w ainsi que plusieurs noms de fichiers. Les options indiquent, respectivement, si l'on veut voir afficher le nombre de lignes, caractères ou mots. L'affichage de chaque résultat sera précédé du nom du fichier concerné.

```

# let l_opt = ref false
  and w_opt = ref false
  and c_opt = ref false ;;
val l_opt : bool ref = {contents=false}
val w_opt : bool ref = {contents=false}
val c_opt : bool ref = {contents=false}

```

Pipeline pour correcteur orthographique

Cet exercice cherche à enchaîner une suite d'actions. Chaque action prend en entrée le résultat de l'action précédente. La communication s'effectue avec des pipes où seule la sortie d'un processus est redirigée vers l'entrée du processus suivant, à la manière du symbole `|` des interprètes de commande Unix.

1. Écrire une fonction [pipe_two_progs](#) de type *string * string list -> string * string list -> unit* telle que `pipe_two_progs (p1,[a1; ...; an]) (p2,[b1; ...; bp])` lance les programmes `p1 a1 ... an` et `p2 b1 ... bp` en redirigeant la sortie standard de `p1` sur l'entrée standard de `p2`. Les `ai` et `bi` sont les arguments de la ligne de commande de chaque programme.

```

# let pipe_two_progs (p1, args1) (p2, args2) =
  let in2, out1 = Unix.pipe() in
  match Unix.fork() with
  0 ->
    Unix.close in2 ;
    Unix.close Unix.stdout ;
    ignore(Unix.dup out1) ;
    Unix.close out1 ;
    Unix.execvp p1 (Array.of_list args1)
  | _ ->
    Unix.close out1 ;
    Unix.close Unix.stdin ;
    ignore(Unix.dup in2) ;
    Unix.close in2 ;
    Unix.execvp p2 (Array.of_list args2) ;;
val pipe_two_progs : string * string list -> string
<fun>

```

2. On reprend la fonction [orthographe](#) de l'exercice de la page ?? pour écrire un premier programme. La modifier pour que la liste des mots incorrects soit envoyée, sans traitement préalable, sous forme d'une ligne par mot sur la sortie standard .

```

# let orthographe dico nom =
  let f = open_in nom in
  try
    while true do
      let s = input_line f in
      let ls = mots s in
      List.iter (Printf.printf"%s\n") (verifie dico ls)
    done ;
    failwith "cas impossible"
  with
  End_of_file -> close_in f
  | x -> close_in f ; raise x ;;
val orthographe : arbre_lex -> string -> unit = <fu

```

3. Le deuxième [programme](#) prend une suite de chaînes de caractères sur son entrée standard, et la trie selon l'ordre lexicographique. On pourra utiliser la fonction `Sort.list` qui trie une liste selon un prédicat donné. La liste triée est ensuite affichée sur la sortie standard.

```
# let trie () =  
  let l = ref [] in  
  try  
    while true do l := Sort.list (<) ((input_line stdi  
    with  
      End_of_file -> List.iter (Printf.printf "%s\n") !l  
val trie : unit -> unit = <fun>
```

4. [Tester](#) la fonction `pipe_two_progs` avec ces deux programmes.

```
# pipe_two_progs ("orthographe", [""; Sys.argv.(1)])
```

5. Écrire une fonction [pipe_n_progs](#) pour enchaîner une liste de programmes.

```
# let dup_stdin in_descr =  
  if in_descr<>Unix.stdin then Unix.dup2 in_descr Un  
  Unix.close in_descr ;;  
val dup_stdin : Unix.file_descr -> unit = <fun>  
  
# let dup_stdout out_descr =  
  if out_descr<>Unix.stdout then Unix.dup2 out_descr  
  Unix.close out_descr ;;  
val dup_stdout : Unix.file_descr -> unit = <fun>
```

6. Écrire un [programme](#) qui supprime les occurrences multiples des éléments d'une liste.

```
# let rmdup () =
  let l = ref [] in
  try
    while true do
      let x = input_line stdin in
      if not (List.mem x !l) then l := x::l
    done
  with End_of_file
  -> List.iter (Printf.printf"%s\n") !l ;;
val rmdup : unit -> unit = <fun>
```

7. [Tester](#) la fonction `pipe_n_progs` avec ces trois programmes.

```
# pipe_n_progs [ ("orthographe", [""; Sys.argv.(1)]);
```

Trace interactive

Lors d'un calcul complexe, il peut être utile d'interagir avec le programme pour vérifier la progression de ce calcul. On reprend pour cela l'exercice de la page [??](#) sur le calcul des nombres premiers contenus dans un intervalle.

1. [Modifier](#) le programme pour qu'une variable globale `result` contienne à tout instant le dernier des nombres premiers déjà trouvés.

```
# let result = ref 0 ;;
val result : int ref = {contents=0}
# let rec eras = function
  [] -> []
  | p::q ->
    result := p ;
    p :: (eras (List.filter (fun x -> x mod p <> 0) q))
val eras : int list -> int list = <fun>
```


2. Écrire une fonction `sigint_handle` pour le traitement du signal `sigint` qui affiche le contenu de `result`.

```
# let sigint_handle (_ : int) =  
  Printf.printf "Current prime number : %d\n" !result  
  flush stdout ;;  
val sigint_handle : int -> unit = <fun>
```

3. **Modifier** le traitement par défaut du signal `sigint` en lui associant la fonction de la question précédente `sigint_handle`.

```
# Sys.set_signal Sys.sigint (Sys.Signal_handle sigint_handle)  
- : unit = ()
```

4. **Compiler** le programme, puis lancer l'exécutable avec une borne supérieure importante pour l'intervalle de recherche. Pendant cette exécution, envoyer le signal `sigint` au processus, soit en utilisant la commande Unix `kill`, soit en tapant le caractère `CTRL-C`.

```
$ ocamlc premiers.ml  
$ premiers 15000  
Current prime number : 2539  
Current prime number : 8263  
2 3 5 7 11 13 17 19 23 29 31 37 41 43 .....
```

Résumé

Ce chapitre a présenté les principales fonctions d'interface avec le système fournies par le module `Unix`. Malgré son nom, ce module offre un grand nombre de fonctions utilisables sous Windows (voir figure 18.1).

En marge de la création de processus, nous nous sommes surtout concentrés sur les possibilités offertes pour communiquer entre plusieurs programmes Objective CAML s'exécutant en même temps

sur la même machine. En particulier, les opérations de bas niveau sur les fichiers, les signaux et les tubes de communication ont été détaillées.

Pour en savoir plus

Le module Unix reprend, jusque dans leurs noms, les fonctions principales des bibliothèques du système Unix. La plupart des paradigmes de programmation utilisés ne sont pas spécifiques à Objective CAML. Le lecteur pourra se référer aux ouvrages classiques sur la programmation système. Citons [Rif90], ou [CDM96] plus spécifique à Linux.

D'autre part, un excellent polycopié de cours de Xavier Leroy [Ler92] ayant pour thème la programmation système en Caml-Light est disponible à l'adresse suivante :

Lien: <http://pauillac.inria.fr/~xleroy/publi/unix-in-caml.ps.gz>

L'implantation du module Unix est l'exemple type de la coopération entre C et Objective CAML. Un grand nombre des fonctions de cette bibliothèque sont de simples appels aux fonctions systèmes C accompagnés du petit travail de transcription des types de données. Les programmes sources de cette implantation sont de bons exemples pour l'interfaçage d'un programme Objective CAML avec une bibliothèque C. Ces programmes sont dans les répertoires `otherlibs/unix` et `otherlibs/win32unix` de la distribution d'Objective CAML.

Ce chapitre a fait l'impasse sur plusieurs fonctionnalités du module Unix. Certains points seront abordés dans le chapitre 20 : prises de communication et adressage Internet. Les autres points, tels la notion de terminal, de système de fichiers, etc, ne sont pas traités dans cet ouvrage. Ils demeurent à découvrir dans l'un des ouvrages cités ci-dessus.

Programmation concurrente

Introduction

On appelle concurrence l'indépendance causale entre plusieurs actions, comme l'exécution de plusieurs instructions en < < même temps > >. C'est la définition que nous donnions du terme « parallèle » en introduction de cette quatrième partie. Les processus de la bibliothèque Unix présentés au chapitre précédent peuvent être qualifiés de concurrents dans la mesure où le système Unix donne l'apparence de la simultanéité de leur exécution sur une machine mono-processeur. Mais la notion de processus et de concurrence ne s'applique pas qu'à ceux obtenus par l'appel système fork.

Le langage Objective CAML possède une bibliothèque pour les processus légers (threads). La principale différence entre un processus léger et un processus engendré par un fork vient du partage ou du non partage de la mémoire entre les différents processus issus du même programme. Entre deux processus légers, seul le contexte d'exécution diffère; le code et la zone mémoire des données sont partagés. Les processus légers n'améliorent pas les temps d'exécution d'une application. Leur principal intérêt est de pouvoir exprimer dans un langage de programmation des algorithmes concurrents.

La nature du langage choisi, impératif ou fonctionnel, fait varier le modèle de concurrence. Pour un programme impératif, comme chaque processus léger peut modifier la mémoire commune, on est dans un modèle à mémoire partagée. La communication entre processus peut se faire par valeurs écrites et lues dans cette mémoire. Pour un programme purement fonctionnel, c'est-à-dire sans effet de bord, et bien que la mémoire soit commune, les calculs qu'exécute chaque processus n'agissent pas sur l'état mémoire partagé. Dans ce cas le modèle utilisé est celui à mémoire distincte et l'interaction entre processus doit s'effectuer par la communication de valeurs via des canaux.

Le langage Objective CAML implante dans sa bibliothèque sur les processus légers les deux modèles. Le module Thread permet de lancer de nouveaux processus correspondants à l'appel d'une fonction avec son argument. Les modules Mutex et Condition apportent les outils de synchronisation que sont les verrous de zone d'exclusion mutuelle et les attentes sur condition. Le module Event

implante un mode de communication de valeurs du langage par événements. Ces valeurs peuvent elles-mêmes être fonctionnelles, permettant ainsi d'échanger entre processus légers des calculs à effectuer. Comme souvent en Objective CAML il est possible de mélanger les deux modèles.

Cette bibliothèque est portable sur les différents systèmes sur lesquels fonctionne Objective CAML. À la différence du module Unix, la bibliothèque Thread permet l'utilisation de processus sur des machines ne tournant pas sous Unix.

Plan du chapitre

La première section précise les interactions possibles entre processus légers, pour ensuite détailler le module Thread et montrer comment exécuter plusieurs processus dans une même application.

La deuxième partie s'intéresse à la synchronisation entre threads par exclusion mutuelle (module Mutex) et avec attente sur condition (module Condition). Deux exemples complets montrent les difficultés inhérentes à ce modèle.

La troisième section explique le mode de communication par événements fourni par le module Event et les nouvelles possibilités qu'il apporte.

La quatrième section conclut ce chapitre par la réalisation d'une file d'attente commune aux différents guichets d'un bureau de poste.

Processus concurrents

L'écriture d'une application composée de plusieurs processus concurrents fait perdre la propriété de déterminisme des programmes séquentiels. Pour des processus partageant une même zone mémoire, le résultat du programme suivant ne peut pas être déduit de sa lecture.

```
programme principal  
lex x = ref 1;;
```

À la fin de l'exécution de P et Q , la référence x peut valoir 2, 3 ou 4, selon l'ordre de calcul de chaque processus.

Cet indéterminisme vaut également pour la terminaison. Comme l'état mémoire dépend du déroulement de chaque processus parallèle, une application peut ne pas terminer pour une certaine exécution et terminer dans une autre. Pour apporter un certain contrôle à l'exécution, les processus doivent se synchroniser.

Pour des processus utilisant des mémoires distinctes, mais communiquant entre eux, leur interaction dépend du type de communication. On introduit pour l'exemple suivant deux primitives de communication : `send` qui envoie une valeur en indiquant le destinataire et `receive` qui reçoit une valeur d'un processus. Soient deux processus communicants P et Q :

processus P	processus Q
<code>lex x = ref 1;;</code>	<code>lex y = ref 1;;</code>
<code>send(Q, !x);</code>	<code>y := !y + 3;</code>
<code>x := !x * 2;</code>	<code>y := !y + receive(P);</code>
<code>send(Q, !x);</code>	<code>send(P, !y);</code>
<code>x := !x + receive(Q);</code>	<code>y := !y + receive(P);</code>

Dans le cas d'une communication évanescente, le processus Q peut rater les émissions de P . On retombe dans le non-déterminisme du modèle précédent.

Pour une communication asynchrone, le médium du canal de communication conserve les différentes valeurs transmises. Seule la réception est bloquante. Le processus P peut être en attente sur Q , bien que ce dernier n'ait pas encore lu les deux envois de P . Ce qui ne l'empêche pas d'émettre.

Pour une communication synchrone, l'émission est elle aussi bloquante. Dans notre exemple le `send(Q, !x)` de P attend que Q soit en réception (`receive(P)`). Une fois l'information transmise, les deux processus continuent leur chemin. Malheureusement, dans notre exemple, P et Q se retrouvent sur une instruction d'émission bloquante, et le programme n'avancera plus.

On peut classer les applications concurrentes en cinq catégories suivant que les unités de programme les composant sont :

1. sans relation ;
2. avec relation mais sans synchronisation ;
3. avec relation d'exclusion mutuelle ;
4. avec relation d'exclusion mutuelle et communication ;
5. avec relation, sans exclusion mutuelle et avec communication synchrone.

La difficulté de réalisation vient principalement des dernières catégories. Nous allons à présent voir comment résoudre ces difficultés en utilisant les bibliothèques d'Objective CAML.

Compilation avec processus légers

La bibliothèque sur les threads d'Objective CAML est découpée en cinq modules dont les quatre premiers définissent chacun des types abstraits :

- module Thread : création et exécution de processus légers (type *Thread.t*);
- module Mutex : création, pose et libération de verrous (type *Mutex.t*);
- module Condition : création de conditions (signaux), attente et réveil sur condition (type *Condition.t*);
- module Event : création de canaux de communication (type *'a Event.channel*), des valeurs y transitant (type *'a Event.event*), et des fonctions de communication.
- module ThreadUnix : redéfinition des fonctions d'entrées-sorties du module Unix pour qu'elles ne soient pas bloquantes.

Cette bibliothèque ne fait pas partie de la bibliothèque d'exécution d'Objective CAML. Son utilisation nécessite soit de compiler ses programmes avec l'option `-custom`, soit de construire une nouvelle boucle d'interaction en utilisant les commandes :

```
$ ocamlc -thread -custom threads.cma fichiers.ml -cclib
$ ocamlmktop -thread -custom -o threadtop thread.cma -co
```

La bibliothèque Threads n'est utilisable avec le compilateur natif que si le système d'exploitation implante des processus légers conformes à la norme POSIX 1003¹. On compile alors les exécutables en ajoutant les bibliothèques `unix.a` et `pthread.a` :

```
$ ocamlc -thread -custom threads.cma fichiers.ml -cclib
-cclib -lunix -cclib -lpthread
$ ocamltop -thread -custom threads.cma fichiers.ml -ccli
-cclib -lunix -cclib -lpthread
$ ocamlcopt -thread threads.cmxa fichiers.ml -cclib -lth
-cclib -lunix -cclib -lpthread
```

Module

Le module `Thread` d'Objective CAML contient les primitives de création et de gestion des processus légers. Nous n'en ferons pas une présentation exhaustive, en particulier les opérations d'entrées-sorties sur les fichiers ont été décrites au chapitre précédent. La création d'un processus léger se fait par appel à :

```
# Thread.create ;;
- : ('a -> 'b) -> 'a -> Thread.t = <fun>
```

Le premier argument, de type `('a -> 'b)`, correspond à la fonction exécutée par le processus créé ; le second argument, de type `'a`, est l'argument attendu par la fonction exécutée ; le résultat de l'appel est le descripteur associé au processus. Le processus ainsi créé est détruit automatiquement lorsque la fonction associée termine.

Connaissant son descripteur, on peut demander l'exécution d'un processus et en attendre la fin en utilisant la fonction `join`. En voici un exemple d'utilisation :

```
# let f_procl () = for i=0 to 10 do Printf.printf "(%d) "
  print_newline() ;;
val f_procl : unit -> unit = <fun>
# let t1 = Thread.create f_procl () ;;
```

```

val t1 : Thread.t = <abstr>
# Thread.join t1 ;;
(0) (1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
- : unit = <unknown constructor>

```

Warning

Le résultat de l'exécution d'un processus n'est pas récupéré par le processus père, mais perdu quand le processus fils termine. On peut également interrompre brutalement le déroulement d'un processus dont on connaît le descripteur par la fonction `kill`. Créons, par exemple, un processus pour l'interrompre immédiatement :

```

# let n = ref 0 ;;
val n : int ref = {contents=0}
# let f_procl () = while true do incr n done ;;
val f_procl : unit -> unit = <fun>
# let go () = n := 0 ;
  let t1 = Thread.create f_procl ()
  in Thread.kill t1 ;
  Printf.printf "n = %d\n" !n ;;
val go : unit -> unit = <fun>
# go () ;;
n = 0
- : unit = ()

```

Un processus peut mettre fin à sa propre activité par la fonction :

```

# Thread.exit ;;
- : unit -> unit = <fun>

```

Il peut suspendre son activité pendant un temps donné par appel à :

```

# Thread.delay ;;
- : float -> unit = <fun>

```

L'argument indique le nombre de secondes d'attente.

Reprenons l'exemple précédent en lui ajoutant des temporisations.

Nous créons un premier processus t1 dont la fonction associée f_proc2 crée à son tour un processus t2 qui exécute f_proc1, puis f_proc2 attend un délai de d secondes, met fin à t2. À la terminaison de t1, on affiche le contenu de n.

```
# let f_proc2 d =  
  n := 0 ;  
  let t2 = Thread.create f_proc1 ()  
  in Thread.delay d ;  
  Thread.kill t2 ;;  
val f_proc2 : float -> unit = <fun>  
# let t1 = Thread.create f_proc2 0.25  
  in Thread.join t1 ; Printf.printf "n = %d\n" !n ;;  
n = 36123  
- : unit = ()
```

Synchronisation des processus

C'est dans le cadre de processus partageant une même zone mémoire que le mot « concurrence » prend tout son sens : les divers processus impliqués sont en concurrence pour l'accès à cette unique ressource qu'est la mémoire². Au problème du partage des ressources vient s'ajouter celui du manque de maîtrise de l'alternance et du temps d'exécution des processus concurrents.

Le système qui gère l'ensemble des processus peut à tout moment interrompre un calcul en cours. Ainsi lorsque deux processus coopèrent, ils doivent pouvoir garantir l'intégrité des manipulations de certaines données partagées. Pour cela, un processus doit pouvoir rester maître de ces données tant qu'il n'a pas achevé un calcul ou toute autre opération (par exemple, une acquisition sur un périphérique). Pour garantir l'exclusivité d'accès aux données à un seul processus, on met en oeuvre un mécanisme dit d'exclusion mutuelle.

Section critique et exclusion mutuelle

Les mécanismes d'exclusion mutuelle sont implantés à l'aide de données particulières appelées verrous. Les opérations sur les

verrous sont limitées à leur création, leur pose et leur libération. Un verrou est la plus petite donnée partagée par un ensemble de processus concurrents. Sa manipulation est toujours exclusive. À la notion d'exclusivité de manipulation d'un verrou s'ajoute celle d'exclusivité de détention : seul le processus qui a pris un verrou peut le libérer ; si d'autres processus veulent à leur tour disposer de ce verrou, ils doivent en attendre la libération par le processus détenteur.

Module

Le module `Mutex` est utilisé pour créer des verrous entre processus en relation d'exclusion mutuelle sur une zone mémoire. Nous allons illustrer leur utilisation par deux petits exemples classiques de concurrence. Les fonctions de création, prise et libération des verrous sont :

```
# Mutex.create ;;
- : unit -> Mutex.t = <fun>
# Mutex.lock ;;
- : Mutex.t -> unit = <fun>
# Mutex.unlock ;;
- : Mutex.t -> unit = <fun>
```

Il existe une variante à la prise de verrou qui n'est pas bloquante :

```
# Mutex.try_lock;;
- : Mutex.t -> bool = <fun>
```

Si le verrou est déjà pris, la fonction retourne **false**. Dans l'autre cas, la fonction prend le verrou et retourne **true**.

Les philosophes orientaux

Cette petite histoire, due à Dijkstra, illustre un pur problème de partage de ressources. Elle se raconte ainsi : < < Cinq philosophes orientaux partagent leur temps entre l'étude et la venue au réfectoire pour manger un bol de riz. La salle affectée à la

sustentation des philosophes ne comprend qu'une seule table ronde sur laquelle sont disposés un grand plat de riz (toujours plein) cinq bols et cinq baguettes. > >

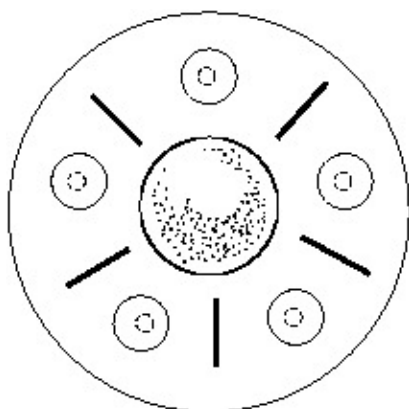


Figure 19.1 : la table des philosophes orientaux

Comme on le voit sur la figure 19.1, un philosophe qui prend les deux baguettes autour de son bol empêche ses voisins d'en faire autant. Dès qu'il dépose l'une de ses baguettes, son voisin, affamé, peut s'en emparer. Le cas échéant, ce dernier devra attendre que l'autre baguette soit disponible. Les baguettes sont ici les ressources partagées.

Pour simplifier les choses, on supposera que chaque philosophe a des habitudes et vient toujours à la même place. On modélise les cinq baguettes par autant de verrous stockés dans un vecteur *b*.

```
# let b =  
  let b0 = Array.create 5 (Mutex.create()) in  
  for i=1 to 4 do b0.(i) <- Mutex.create() done;  
  b0 ;;  
val b : Mutex.t array = [|<abstr>; <abstr>; <abstr>; <abstr>; <abstr>|]
```

Manger et méditer sont simulés par une suspension des processus.

```
# let mediter = Thread.delay
  and manger = Thread.delay ;;
val mediter : float -> unit = <fun>
val manger : float -> unit = <fun>
```

On modélise un philosophe par une fonction qui exécute à l'infini la séquence des actions de la petite histoire de Dijkstra. La prise d'une baguette est simulée par l'obtention d'un verrou, ainsi un seul philosophe à la fois peut détenir une baguette. On introduit un petit temps de réflexion entre la prise et le dépôt de chacune des deux baguettes ainsi que quelques affichages traçant l'activité du philosophe.

```
# let philosophe i =
  let ii = (i+1) mod 5
  in while true do
    mediter 3. ;
    Mutex.lock b.(i);
    Printf.printf "Le philosophe (%d) prend sa baguette gauche\n";
    Printf.printf " et médite encore un peu\n";
    mediter 0.2;
    Mutex.lock b.(ii);
    Printf.printf "Le philosophe (%d) prend sa baguette droite\n";
    manger 0.5;
    Mutex.unlock b.(i);
    Printf.printf "Le philosophe (%d) repose sa baguette gauche\n";
    Printf.printf " et commence déjà à méditer\n";
    mediter 0.15;
    Mutex.unlock b.(ii);
    Printf.printf "Le philosophe (%d) repose sa baguette droite\n";
  done ;;
val philosophe : int -> unit = <fun>
```

On pourra tester ce petit programme en exécutant :

```
for i=0 to 4 do ignore (Thread.create philosophe i) done
while true do Thread.delay 5. done ;;
```

On suspend, dans la boucle infinie **while**, le processus principal de façon à augmenter les chances des processus philosophes d'entrer en action. On utilise dans la boucle d'activité des philosophes des

délais d'attente dont la durée est choisie aléatoirement dans le but de créer un peu de disparité dans l'exécution parallèle des processus.

Problèmes de la solution naïve

Il peut arriver une chose terrible à nos philosophes : ils arrivent tous en même temps et se saisissent de leur baguette gauche. Dans ce cas nous sommes dans une situation d'inter-blocage. Plus aucun philosophe ne pourra manger ! Nous sommes dans un cas de famine.

Pour éviter cela, les philosophes peuvent reposer une baguette s'il n'arrivent pas à prendre la deuxième. Cela est fort courtois, mais permet à deux philosophes de se liguer contre un troisième pour l'empêcher de se sustenter en ne relâchant les baguettes que si l'autre voisin les a prises. Il existe diverses solutions à ce problème. L'une d'elle fait l'objet de l'exercice page ??.

Producteurs et consommateurs I

Le couple producteurs-consommateurs est un exemple classique de la programmation concurrente. Un groupe de processus, désignés comme les producteurs, est chargé d'emmagasiner des données dans une file d'attente ; un second groupe, les consommateurs, est chargé de les déstocker. Chaque intervenant exclut les autres.

Nous implantons ce schéma en utilisant une file d'attente partagée entre les producteurs et les consommateurs. Pour garantir le bon fonctionnement de l'ensemble, la file d'attente est manipulée en exclusion mutuelle afin de garantir l'intégrité des opérations d'ajout et de retrait. `f` est la file d'attente partagée, et `m` le verrou d'exclusion mutuelle.

```
# let f = Queue.create () and m = Mutex.create () ;;  
val f : '_a Queue.t = <abstr>  
val m : Mutex.t = <abstr>
```

Nous divisons l'activité d'un producteur en deux parties : créer un

produit (fonction produire) et stocker un produit (fonction stocker).
Seule l'opération de stockage a besoin du verrou.

```
# let produire i p d =  
  incr p ;  
  Thread.delay d ;  
  Printf.printf "Le producteur (%d) a produit %d\n" i !p  
  flush stdout ;;  
val produire : int -> int ref -> float -> unit = <fun>  
  
# let stocker i p =  
  Mutex.lock m ;  
  Queue.add (i,!p) f ;  
  Printf.printf "Le producteur (%d) a ajouté son %d-ième  
  flush stdout ;  
  Mutex.unlock m ;;  
val stocker : int -> int ref -> unit = <fun>
```

Le code du producteur est une boucle sans fin de création et de stockage. Nous introduisons un délai aléatoire d'attente à la fin de chaque itération afin d'obtenir un effet de désynchronisation dans l'exécution.

```
# let producteur i =  
  let p = ref 0 and d = Random.float 2.  
  in while true do  
    produire i p d ;  
    stocker i p ;  
    Thread.delay (Random.float 2.5)  
  done ;;  
val producteur : int -> unit = <fun>
```

La seule opération du consommateur est le retrait d'un élément de la file en prenant garde à l'existence effective d'un produit.

```
# let consommateur i =  
  while true do  
    Mutex.lock m ;  
    ( try  
      let ip, p = Queue.take f  
    in Printf.printf "Le consommateur(%d) " i ;
```

```

Printf.printf "a retiré le produit (%d,%d)\n" ip p ;
flush stdout ;
with
Queue.Empty ->
Printf.printf "Le consommateur(%d) " i ;
print_string "est reparti les mains vides\n" ) ;
Mutex.unlock m ;
Thread.delay (Random.float 2.5)
done ;;
val consommateur : int -> unit = <fun>

```

Le programme de test suivant crée quatre producteurs et quatre consommateurs.

```

for i = 0 to 3 do
  ignore (Thread.create producteur i);
  ignore (Thread.create consommateur i)
done ;
while true do Thread.delay 5. done ;;

```

Attente et synchronisation

La relation d'exclusion mutuelle n'est pas assez fine pour décrire la synchronisation entre processus. Il n'est pas rare que le travail d'un processus dépende de la réalisation d'une action par un autre processus, modifiant par là une certaine condition. Il est alors souhaitable que les processus puissent communiquer le fait que cette condition a pu changer, indiquant aux processus en attente de la tester de nouveau. Les différents processus sont alors en relation d'exclusion mutuelle avec communication.

Dans l'exemple précédent, un consommateur, plutôt que repartir les mains vides pourrait attendre qu'un producteur vienne réapprovisionner le stock. Ce dernier pourra signaler au consommateur en attente qu'il y a quelque chose à emporter. Le modèle de l'attente sur condition d'une prise de verrou est connu sous le nom de sémaphore.

Sémaphores

Un sémaphore est une variable entière s ne pouvant prendre que des valeurs positives (ou nulles). Une fois s initialisée, les seules opérations admises sont : $wait(s)$ et $signal(s)$, notées respectivement $P(s)$ et $V(s)$. Elles sont définies ainsi, s correspond au nombre de ressources d'un type donné.

- $wait(s)$: si $s > 0$ alors $s := s - 1$, sinon l'exécution du processus ayant appelé $wait(s)$ est suspendue.
- $signal(s)$: si un processus a été suspendu lors d'une exécution antérieure d'un $wait(s)$ alors le réveiller, sinon $s := s + 1$.

Un sémaphore ne prenant que les valeurs 0 ou 1 est appelé sémaphore binaire.

Module

Les fonctions du module Condition implantent des primitives de mise en sommeil et de réveil de processus sur un signal. Un signal, dans ce cadre, est une variable partagée par l'ensemble des processus. Son type est abstrait et les fonctions de manipulation sont :

create

: $unit \rightarrow Condition.t$ qui crée un nouveau signal.

signal

: $Condition.t \rightarrow unit$ qui réveille l'un des processus en attente du signal.

broadcast

: $Condition.t \rightarrow unit$ qui réveille l'ensemble des processus en attente du signal.

wait

: $Condition.t \rightarrow Mutex.t \rightarrow unit$ qui suspend le processus appelant sur le signal passé en premier argument. Le second argument est un verrou utilisé pour protéger la manipulation du signal. Il est libéré, puis repositionné à chaque exécution de la fonction.

Producteurs et consommateurs (2)

Nous reprenons l'exemple des producteurs et consommateurs en utilisant le mécanisme des variables de condition pour mettre en attente un consommateur arrivant alors que le magasin est vide.

Pour réaliser la synchronisation entre attente d'un consommateur et production, on déclare :

```
# let c = Condition.create () ;;  
val c : Condition.t = <abstr>
```

On modifie la fonction de stockage du producteur en lui ajoutant l'émission d'un signal :

```
# let stocker2 i p =  
  Mutex.lock m ;  
  Queue.add (i,!p) f ;  
  Printf.printf "Le producteur (%d) a ajoute son %d-ième  
  flush stdout ;  
  Condition.signal c ;  
  Mutex.unlock m ;;  
val stocker2 : int -> int ref -> unit = <fun>  
# let producteur2 i =  
  let p = ref 0 in  
  let d = Random.float 2.  
  in while true do  
    produire i p d;  
    stocker2 i p;  
    Thread.delay (Random.float 2.5)  
  done ;;  
val producteur2 : int -> unit = <fun>
```

L'activité du consommateur se fait alors en deux temps : attendre qu'un produit soit disponible puis emporter le produit. Le verrou est pris quand l'attente prend fin et il est rendu dès que le consommateur a emporté son produit. L'attente se fait sur la variable c.

```
# let attendre2 i =
```

```

Mutex.lock m ;
while Queue.length f = 0 do
Printf.printf "Le consommateur (%d) attend\n" i ;
Condition.wait c m
done ;;
val attendre2 : int -> unit = <fun>
# let emporter2 i =
  let ip, p = Queue.take f in
  Printf.printf "Le consommateur (%d) " i ;
  Printf.printf "emporte le produit (%d, %d)\n" ip p ;
  flush stdout ;
  Mutex.unlock m ;;
val emporter2 : int -> unit = <fun>
# let consommateur2 i =
  while true do
    attendre2 i;
    emporter2 i;
    Thread.delay (Random.float 2.5)
  done ;;
val consommateur2 : int -> unit = <fun>

```

Notons qu'il n'est plus besoin de vérifier l'existence effective d'un produit puisqu'un consommateur doit attendre l'existence d'un produit dans la file d'attente pour que sa suspension prenne fin. Vu que la fin de son attente correspond à la prise du verrou, il ne court pas le risque de se faire dérober le nouveau produit avant de l'avoir emporté.

Lecteurs et écrivain

Voici un autre exemple classique de processus concurrents dans lequel les agents n'ont pas le même comportement vis-à-vis de la donnée partagée.

Un écrivain et des lecteurs opèrent sur une donnée partagée. L'action du premier est susceptible de rendre momentanément les données incohérentes alors que les seconds n'ont qu'une action passive. La difficulté vient du fait que nous ne souhaitons pas

interdire à plusieurs lecteurs de consulter la donnée simultanément. Une solution à ce problème est de gérer un compteur dénombrant les lecteurs en train d'accéder aux données. L'écriture ne sera acceptable que si le nombre de lecteurs est nul.

La donnée est symbolisée par l'entier *data* qui prendra la valeur *0* ou *1*. La valeur *0* signifie que la donnée est prête pour la lecture :

```
# let data = ref 0 ;;
val data : int ref = {contents=0}
```

Les opérations sur le compteur *n* sont protégées par le verrou *m* :

```
# let n = ref 0 ;;
val n : int ref = {contents=0}
# let m = Mutex.create () ;;
val m : Mutex.t = <abstr>
# let cpt_incr () = Mutex.lock m ; incr n ; Mutex.unlock m
val cpt_incr : unit -> unit = <fun>
# let cpt_decr () = Mutex.lock m ; decr n ; Mutex.unlock m
val cpt_decr : unit -> unit = <fun>
# let cpt_signal () = Mutex.lock m ;
  if !n=0 then Condition.signal c ;
  Mutex.unlock m ;;
val cpt_signal : unit -> unit = <fun>
```

Les lecteurs mettent à jour le compteur et émettent le signal *c* lorsque plus aucun lecteur n'est présent. Ils indiquent par là à l'écrivain qu'il peut entrer en action.

```
# let c = Condition.create () ;;
val c : Condition.t = <abstr>
# let lire i =
  cpt_incr () ;
  Printf.printf "Le lecteur (%d) lit (donnée=%d)\n" i !data
  Thread.delay (Random.float 1.5) ;
  Printf.printf "Le lecteur (%d) a fini sa lecture\n" i ;
  cpt_decr () ;
  cpt_signal () ;;
val lire : int -> unit = <fun>
```

```
# let lecteur i = while true do lire i ; Thread.delay (R
val lecteur : int -> unit = <fun>
```

L'écrivain tente de bloquer le compteur pour interdire aux lecteurs d'accéder à la donnée partagée. Mais il ne peut le faire que si le compteur est nul, sinon il attend le signal lui indiquant que c'est le cas.

```
# let ecrire () =
  Mutex.lock m ;
  while !n<>0 do Condition.wait c m done ;
  print_string "L'écrivain écrit\n" ; flush stdout ;
  data := 1 ; Thread.delay (Random.float 1.) ; data := 0
  Mutex.unlock m ;;
val ecrire : unit -> unit = <fun>
```

```
# let ecrivain () =
  while true do ecrire () ; Thread.delay (Random.float 1.
val ecrivain : unit -> unit = <fun>
```

On crée pour tester ces fonctions un écrivain et six lecteurs.

```
ignore (Thread.create ecrivain ());
for i=0 to 5 do ignore(Thread.create lecteur i) done;
while true do Thread.delay 5. done ;;
```

Cette solution garantit que ni l'écrivain ni les lecteurs n'ont accès aux données en même temps. Par contre, rien ne garantit que l'écrivain puisse un jour remplir son office, nous sommes confrontés là encore à un cas de famine.

Communication synchrone

Le module Event de la bibliothèque des processus légers plante la communication de valeurs quelconques entre deux processus via des canaux de communication particuliers. La communication effective de la valeur est synchronisée au moyen d'événements d'émission ou de réception.

Ce modèle de communication synchronisée sur des événements

permet de transférer sur des canaux typés des valeurs du langage, y compris des fermetures, des objets ou des événements. Il est décrit dans [[Rep92](#)].

Synchronisation sur événements de communication

Les événements primitifs de communication sont :

- `send c v` envoie une valeur `v` sur le canal `c`
- `receive c` réceptionne une valeur sur la canal `c`

Pour qu'ils réalisent l'action physique qui leur est associée, deux événements doivent être synchronisés. Pour cela on introduit une opération de synchronisation (`sync`) sur les événements. L'émission et la réception d'une valeur ne sont effectives que lorsque les deux processus communicants sont en phase. Si un seul processus cherche à se synchroniser, l'opération est bloquante et attend que le deuxième processus effectue sa synchronisation. Cela implique qu'un émetteur cherchant à synchroniser l'envoi d'une valeur (`sync (send c v)`) peut se retrouver bloqué en attente d'une synchronisation du récepteur (`sync (receive c)`).

Valeurs transmises

Les canaux de communication par où transitent les valeurs échangées sont typés : seules les valeurs du type paramètre du canal `y` sont acceptées. Rien n'empêche de créer de nombreux canaux par type de valeur à communiquer. Comme cette communication s'effectue entre processus légers Objective CAML, n'importe quelle valeur du langage peut être envoyée sur un canal de son type. Cela est valable pour les fermetures, les objets et aussi des événements pour une demande de synchronisation déportée.

Module

Les valeurs encapsulées dans les événements de communication transitent par des canaux de communication du type abstrait *'a channel*. La fonction de création d'un canal est :

```
# Event.new_channel ;;
- : unit -> 'a Event.channel = <fun>
```

Les événements d'émission et de réception sont créés par un appel aux fonctions :

```
# Event.send ;;
- : 'a Event.channel -> 'a -> unit Event.event = <fun>
# Event.receive ;;
- : 'a Event.channel -> 'a Event.event = <fun>
```

On peut considérer les fonctions `send` et `receive` comme des constructeurs du type abstrait *'a event*. L'événement construit à partir de l'émission ne conserve pas une information de type de la valeur à transmettre (type *unit Event.event*). Par contre l'événement de réception en tient compte pour récupérer la valeur pendant une synchronisation. Ces fonctions ne sont pas bloquantes dans la mesure où la transmission d'une valeur n'est réalisée qu'au moment de la synchronisation de deux processus par la fonction :

```
# Event.sync ;;
- : 'a Event.event -> 'a = <fun>
```

Cette fonction peut être est bloquante pour l'émetteur et le récepteur.

Il en existe une version non bloquante :

```
# Event.poll ;;
- : 'a Event.event -> 'a option = <fun>
```

Cette fonction vérifie qu'un autre processus est en attente de synchronisation. Si c'est le cas elle réalise les transmissions, et retourne la valeur `Some v` si `v` est la valeur associée à l'événement, et `None` sinon. Le message reçu, extrait par la fonction `sync` peut être le résultat d'un processus plus ou moins compliqué mettant en oeuvre d'autres échanges de messages.

Exemple de synchronisation

On définit trois processus légers, le premier t1 envoie une chaîne de caractères sur le canal c (fonction g) partagé par tous les processus. Les deux autres t2 et t3 attendent une valeur sur ce même canal. Voici les fonctions exécutées par les différents processus :

```
# let c = Event.new_channel ();;
val c : '_a Event.channel = <abstr>
# let f () =
  let ids = string_of_int (Thread.id (Thread.self ()))
  in print_string ("----- avant -----" ^ ids) ; pri
  let e = Event.receive c
  in print_string ("----- pendant -----" ^ ids) ; p
  let v = Event.sync e
  in print_string (v ^ " " ^ ids ^ " ") ;
  print_string ("----- après -----" ^ ids) ; print_
val f : unit -> unit = <fun>
# let g () =
  let ids = string_of_int (Thread.id (Thread.self ()))
  in print_string ("Début de " ^ ids ^ "\n");
  let e2 = Event.send c "hello"
  in Event.sync e2 ;
  print_string ("Fin de " ^ ids) ;
  print_newline () ;;
val g : unit -> unit = <fun>
```

Les trois processus sont alors créés et exécutés :

```
# let t1,t2,t3 = Thread.create f (), Thread.create f (),
val t1 : Thread.t = <abstr>
val t2 : Thread.t = <abstr>
val t3 : Thread.t = <abstr>
# Thread.delay 1.0;;
Début de 5
----- avant -----6
----- pendant -----6
hello 6 ----- après -----6
----- avant -----7
----- pendant -----7
Fin de 5
- : unit = <unknown constructor>
```

L'émission peut être bloquante. La trace de fin de t1 s'affiche après les traces de synchronisation de t2 et t3. Seul un des deux processus t1 ou t2 est vraiment terminé, comme le montrent les appels suivants :

```
# Thread.kill t1;;  
- : unit = ()  
# Thread.kill t2;;  
Uncaught exception: Failure("Thread.kill: killed thread")
```

Exemple : bureau de Poste

Nous présentons, pour finir ce chapitre, un exemple un peu plus complet de programme concurrent : la modélisation d'une file d'attente commune aux divers guichets d'un bureau de poste.

Comme souvent en programmation concurrente les problèmes sont posés métaphoriquement, mais remplacez les guichets du bureau de poste par un ensemble d'imprimantes et vous avez la solution d'un vrai problème d'informatique.

Voici la politique de gestion de service que nous proposons, elle n'a rien d'originale mais a fait ses preuves : chaque client prend en arrivant un numéro d'attente ; lorsqu'un guichetier a fini de servir un client, il appelle un numéro. Lorsque son numéro est appelé, le client se rend au guichet correspondant.

Organisation du développement

Nous distinguerons dans notre développement les *ressources* et les *agents*. Les premières sont : le distributeur de numéros d'attente, l'afficheur d'appel et les guichets. Les seconds sont : les guichetiers et les clients. Les ressources sont modélisées par des objets. Les agents le sont par des fonctions exécutées par un processus léger.

Les ressources sont les données partagées par les différents agents du système. L'encapsulation de leurs données et actions dans un objet permet de centraliser, pour chacune, la gestion des mécanismes d'exclusion mutuelle. Lorsqu'un agent désire modifier

ou consulter l'état d'un objet, il n'a pas à connaître ni à gérer lui-même les verrous, ce qui permet de simplifier l'organisation de l'accès aux données sensibles et d'éviter des oublis lors du codage des agents.

Le matériel

Le distributeur

Le distributeur de numéros d'attente comprend deux champs de données : un compteur et un verrou. La seule méthode offerte par le distributeur est la prise d'un nouveau numéro.

```
# class distrib () =  
  object  
    val mutable n = 0  
    val m = Mutex.create()  
    method prendre () = let r = Mutex.lock m ; n <- n+1 ; r  
    in Mutex.unlock m ; r  
  end ;;  
class distrib :  
  unit ->  
  object val m : Mutex.t val mutable n : int method prend
```

Le verrou interdit que deux clients prennent en même temps un numéro. Notez la façon dont on utilise une variable intermédiaire (r) pour garantir que le numéro calculé dans la section critique est bien celui retourné par l'appel à la méthode.

L'afficheur

L'afficheur des numéros appelés contient trois champs de données : un entier (le numéro appelé) ; un verrou et une variable de condition. Les deux méthodes sont : consultation du numéro appelé (attendre) et modification (appeler).

```
# class affich () =
```

```

object
val mutable nc = 0
val m = Mutex.create()
val c = Condition.create()

method attendre n =
  Mutex.lock m;
  while n > nc do Condition.wait c m done;
  Mutex.unlock m;

method appel () =
  let r = Mutex.lock m ;
  nc <- nc+1 ;
  nc
  in Condition.broadcast c ;
  Mutex.unlock m ;
  r
  end ;;

```

La variable de condition est utilisée pour mettre les clients en sommeil sur l'attente de leur numéro. Ils sont tous réveillés lorsque la méthode appel est invoquée. L'accès en lecture ou écriture du numéro appelé est protégé par un même verrou.

Le guichet

Le guichet comprend cinq champs de données : un numéro de guichet fixe (variable ng) ; le numéro du client attendu (variable nc) ; un booléen (variable libre) ; un verrou et une variable de condition. Il offre huit méthodes, dont deux privées : deux méthodes d'accès simple (méthodes get_ng et get_nc) ; un groupe de trois méthodes simulant l'attente du guichetier entre deux clients (méthode privée attendre et méthodes publiques attendre_arrivee, attendre_depart) ; un groupe de trois méthodes simulant l'occupation du guichet (méthode privée set_libre et méthodes arriver, partir).

```

# class guichet (i:int) =

```

```

object (self)
val ng = i
val mutable nc = 0
val mutable libre = true
val m = Mutex.create()
val c = Condition.create()

method get_ng = ng
method get_nc = nc

method private attendre f =
  Mutex.lock m ;
  while f () do Condition.wait c m done ;
  Mutex.unlock m

method attendre_arrivee n = nc <- n ; self#attendre (f)
method attendre_depart () = self#attendre (fun () -> n <- n - 1)

method private set_libre b =
  Mutex.lock m ;
  libre <- b ;
  Condition.signal c ;
  Mutex.unlock m
method arriver () = self#set_libre false
method partir () = self#set_libre true

end ;;

```

Un bureau de poste

On regroupe ces trois ressources dans un enregistrement :

```

# type bureau = { d : distrib ; a : affich ; gs : guichet }

```

Clients et guichetiers

Le comportement de l'ensemble du système dépendra des trois paramètres suivants :

```
# let delai_service = 1.7 ;;
# let delai_arrivee = 1.7 ;;
# let delai_guichet = 0.5 ;;
```

Ils représentent chacun la valeur maximale du paramètre dont la valeur effective sera tirée au hasard dans la limite des bornes fixées. Le premier paramètre modélise le temps de service d'un client ; le deuxième le délai d'arrivée des clients dans le bureau de poste ; le dernier le temps que met un guichetier pour appeler un nouveau client après que le dernier soit parti.

Le guichetier

Le travail d'un guichetier consiste à boucler indéfiniment sur la séquence suivante :

1. Appeler un numéro.
2. Attendre l'arrivée du client porteur du numéro appelé.
3. Attendre le départ du client occupant son guichet.

En rajoutant quelques affichages, nous obtenons la fonction :

```
# let guichetier ((a:affich), (g:guichet)) =
  while true do
    let n = a#appel ()
    in Printf.printf"Guichet %d appelle %d\n" g#get_ng n ;
    g#attendre_arrivee n ;
    g#attendre_depart () ;
    Thread.delay (Random.float delai_guichet)
  done ;;
val guichetier : affich * guichet -> unit = <fun>
```

Le client

Un client exécute la séquence suivante :

1. Prendre un numéro d'attente.
2. Attendre que son numéro soit appelé.
3. Se rendre au guichet ayant appelé son numéro pour se faire servir.

La seule activité un peu complexe du client est de chercher le guichet où il est attendu.

On se donne, pour ce, la fonction auxiliaire :

```
# let chercher_guichet n gs =
  let i = ref 0 in while gs.(!i)#get_nc <> n do incr i
val chercher_guichet : 'a -> < get_nc : 'a; .. > array -
```

En rajoutant quelques affichages, la fonction principale du client est :

```
# let client b =
  let n = b.d#prendre()
  in Printf.printf "Arrivée client %d\n" n ; flush stdout
  b.a#attendre n ;
  let ig = chercher_guichet n b.gs
  in b.gs.(ig)#arriver () ;
  Printf.printf "Le client %d occupe le guichet %d\n" n i
  flush stdout ;
  Thread.delay (Random.float delai_service) ;
  b.gs.(ig)#partir () ;
  Printf.printf "Le client %d s'en va\n" n ; flush stdout
val client : bureau -> unit = <fun>
```

L'ensemble

Le programme principal de l'application crée un bureau de poste et ses guichetiers (chaque guichetier est un processus) puis lance un processus chargé d'engendrer indéfiniment des clients (chaque client est aussi un processus).

```
# let main () =
```

```

let b =
{ d = new distrib();
a = new affich();
gs = (let gs0 = Array.create 5 (new guichet 0) in
for i=0 to 4 do gs0.(i) <- new guichet i done;
gs0)
}
in for i=0 to 4 do ignore (Thread.create guichetier (b.
let creer_clients b = while true do
ignore (Thread.create client b) ;
Thread.delay (Random.float delai_arrivee)
done
in ignore (Thread.create creer_clients b) ;
Thread.sleep () ;;
val main : unit -> unit = <fun>

```

La dernière instruction endort le processus associé au programme afin de passer tout de suite la main aux autres processus actifs de l'application.

Exercices

Les philosophes débloqués

Pour résoudre l'éventuel blocage des philosophes orientaux, il suffit de limiter à quatre l'accès simultané à la table. Implantez cette [solution](#)

On rajoute un compteur indiquant le nombre de philosophes présents ainsi que deux fonctions, entrer et sortir, chargées respectivement de limiter les entrées et signaler les sorties.

```

# let entrer, sortir =
  let n = ref 0 in
  let m = Mutex.create() in
  let c = Condition.create() in

```

```

let loc_entrer () =
  Mutex.lock m ;
  while not (!n < 4) do Condition.wait c m done ;
  incr n ;
  if !n > 1 then Printf.printf "%d philosophes sont a
else Printf.printf "%d philosophe est a table\n" !n
  flush stdout;
  Mutex.unlock m
in
let loc_sortir () =
  Mutex.lock m ;
  decr n ;
  Mutex.unlock m ;
  Condition.broadcast c
in
  loc_entrer, loc_sortir ;;
val entrer : unit -> unit = <fun>
val sortir : unit -> unit = <fun>

```

Il suffit alors d'appeler ces fonctions en début et en fin de boucle d'un philosophe.

```

# let philosophe i =
  let ii = (i+1) mod 4 in
  while true do
    Printf.printf "Le philosophe (%d) se pointe\n" i ;
    entrer () ;
    mediter 3. ;
    Mutex.lock b.(i);
    Printf.printf
      "Le philosophe (%d) prend sa baguette gauche et medite
    mediter 0.2; Mutex.lock b.(ii) ;
    Printf.printf "Le philosophe (%d) prend sa baguette droi
    manger 0.5 ;
    Mutex.unlock b.(i) ;
    Printf.printf
      "Le philosophe (%d) rend sa baguette gauche et commence
    mediter 0.15 ;
    Mutex.unlock b.(ii) ;
    Printf.printf "Le philosophe (%d) rend sa baguette droi

```

```

sortir () ;
Printf.printf "Le philosophe (%d) se tire\n" i ;
done ;;
val philosophe : int -> unit = <fun>

```

Attention, cette solution supprime les inter-blocages, mais pas les famines. Pour résoudre ce dernier problème, on peut soit se fier au hasard en introduisant un délai d'attente en après la sortie d'un philosophe, soit gérer explicitement une file d'attente.

La poste en fait plus

Nous proposons l'amendement suivant au bureau de poste décrit à la page ?? : certains clients impatientes peuvent partir avant que leur numéro ne soit appelé.

1. Ajouter à la classe **distrib** une méthode attendre (de type *int -> unit*) qui met en attente l'appelant tant que le dernier numéro distribué est inférieur ou égal au paramètre de la méthode (il faut modifier prendre de façon à ce qu'elle émette un signal).

```

# class distrib () =
  object
    val mutable n = 0
    val m = Mutex.create ()
    val c = Condition.create ()

    method attendre nc =
      Mutex.lock m ;
      while (n <= nc) do Condition.wait c m done ;
      Mutex.unlock m

    method prendre () =
      Mutex.lock m ;
      n <- n+1 ;
      let nn = n in
      Condition.broadcast c ;
      Mutex.unlock m ;

```



```

nn
end ;;
class distrib :
  unit ->
  object
  val c : Condition.t
  val m : Mutex.t
  val mutable n : int
  method attendre : int -> unit
  method prendre : unit -> int
end

```

2. Modifier la méthode `attendre_arrivee` de la classe `guichet` de façon à ce qu'elle retourne le booléen `true` si le client attendu s'est présenté et `false` si le client ne s'est pas présenté au bout d'un certain temps.

```

#
method private reveil t =
  let dt = delai_attente_appel /. 10.0 in
  while (Unix.gettimeofday() < t) do Thread.delay dt
  Condition.signal c

method attendre_arrivee () =
  let t = Unix.gettimeofday() +. delai_attente_appel
  let r = Thread.create self#reveil t in
  Mutex.lock m;
  while libre && (Unix.gettimeofday() < t) do
    Condition.wait c m
  done;
  (try Thread.kill r with _ -> ());
  let b = not libre in (Mutex.unlock m; b)

```

3. Modifier la classe `affiche` en lui passant en paramètre un distributeur de numéros et en :

1. ajoutant une méthode `attendre_jusqu'a` qui renvoie `true` si le numéro attendu a été appelé pendant un délai

d'attente donné en paramètre et false sinon ;

```
# class affich (d:distrib) =
  object
    val mutable nc = 0
    val m = Mutex.create ()
    val c = Condition.create ()

    method attendre n =
      Mutex.lock m ;
      while nc < n do Condition.wait c m done ;
      Mutex.unlock m

    method attendre_jusqu'a n t =
      Mutex.lock m ;
      while (nc < n) && (Unix.gettimeofday() < t) do
        let b = not (nc < n) in
          Mutex.unlock m ;
          b

    method appel (g:guichet) =
      Mutex.lock m ;
      d#attendre nc ;
      nc <- nc+1 ;
      g#set_nc nc ;
      Condition.broadcast c ;
      Mutex.unlock m

  end ;;
class affich :
  distrib ->
  object
    val c : Condition.t
    val m : Mutex.t
    val mutable nc : int
    method appel : guichet -> unit
    method attendre : int -> unit
    method attendre_jusqu'a : int -> float -> bool
  end
```

2. modifiant la méthode `appel` pour qu'elle prenne en paramètre un guichet et mette à jour le champ `nc` de ce guichet (il faut rajouter une méthode de mise à jour dans la classe `guichet`).

4. Modifier la fonction `guichetier` pour tenir compte des attentes infructueuses.

```
# type bureau = { d: distrib; a: affich; gs: guiche
val delai_service : float = 4
val delai_arrivee : float = 2
val delai_guichet : float = 0.5
val delai_attente_client : float = 0.7

let guichetier ((a : affich), (g : guichet)) =
  while true do
    a#appel g ;
    Printf.printf"Guichet %d appelle %d\n" g#get_nc g#
    if g#attendre_arrivee () then g#attendre_depart ()
    else
      begin
        Printf.printf"Guichet %d n'attend plus %d\n" g#get_
        flush stdout
      end ;
      Thread.delay (Random.float delai_guichet)
    done ;;
val guichetier : affich * guichet -> unit = <fun>
```

5. Écrire une fonction `client_impatient` qui simule le comportement d'un client impatient.

```
# val chercher_guichet : 'a -> < get_nc : 'a; .. >

let client_impatient b =
  let n = b.d#prendre() in
  let t = Unix.gettimeofday() +. (Random.float delai
```

```

Printf.printf "Arrivee client impatient %d\n" n; f
if b.a#attendre_jusqu'a n t
then
  let ig = chercher_guichet n b.gs in
  b.gs.(ig)#arriver() ;
  Printf.printf "Le client %d occupe le guichet %d\n"
  flush stdout ;
  Thread.delay (Random.float delai_service) ;
  b.gs.(ig)#partir() ;
  Printf.printf "Le client %d s'en va\n" n
else
  Printf.printf "Le client %d, las d'attendre, s'en
  flush stdout ;;
Characters 518-531:
This function is applied to too many arguments

```

Producteurs et consommateurs d'objet

Cet exercice reprend l'algorithme des producteurs-consommateurs avec la variante suivante : le magasin de stockage est de taille finie (i.e. un tableau plutôt qu'une liste gérée en FIFO). De plus, nous proposons d'en faire une implantation utilisant, à l'instar du bureau de poste, les objets pour modéliser les ressources.

1. Définir une classe **produit** de signature :

```

class produit : string ->
  object
    val nom : string
    method nom : string
  end

# class produit (s:string) =
  object

```

```

val nom = s
method nom = nom
end ;;
class produit : string -> object val nom : string m

```

2. Définir une classe **magasin** telle que :

```

class magasin : int ->
  object
    val mutable buffer : produit array
    val c : Condition.t
    val mutable ic : int
    val mutable ip : int
    val m : Mutex.t
    val mutable np : int
    val taille : int
    method déposer : produit -> unit
    method prendre : unit -> produit
  end

```

Les indices ic et ip sont manipulés respectivement par les producteurs et les consommateurs. L'indice ic donne l'indice du dernier produit pris et ip celui du dernier produit stocké. Le compteur np donne le nombre de produits en stock. L'exclusion mutuelle et la mise en attente des producteurs et des consommateurs seront gérées par les méthodes de cette classe.

```

# class magasin n =
  object (self)
    val mutable taille = n;
    val mutable np = 0
    val mutable buffer = ([||] : produit array)
    val mutable ip = 0 (* Indice producteur *)
    val mutable ic = 0 (* Indice consommateur *)

    val m = Mutex.create ()
    val c = Condition.create ()

```

```

initializer buffer<-Array.create n (new produit "e

method display1 () =
let i = ip mod taille in
Printf.printf "Ajout (%d)%s\n" i ((buffer.(i))#nom

method deposer p =
Mutex.lock m ;
while (ip-ic+1 > Array.length(buffer)) do Condition
buffer.(ip mod taille) <- p ;
self#display1() ;
ip <- ip+1 ;
Mutex.unlock m ;
Condition.signal c

method display2 () =
let i = ic mod taille in
Printf.printf "Retrait (%d)%s\n" i ((buffer.(i))#n

method prendre () =
Mutex.lock m ;
while(ip == ic) do Condition.wait c m done ;
self#display2() ;
let r = buffer.(ic mod taille) in
ic<- ic+1 ;
Mutex.unlock m ;
Condition.signal c ;
r
end ;;
class magasin :
int ->
object
val mutable buffer : produit array
val c : Condition.t
val mutable ic : int
val mutable ip : int
val m : Mutex.t
val mutable np : int
val mutable taille : int

```

```

method depoter : produit -> unit
method display1 : unit -> unit
method display2 : unit -> unit
method prendre : unit -> produit
end

```

3. Définir une fonction **consommateur** : magasin -> string -> unit.

```

# let consommateur mag na =
  while true do
    let p = mag#prendre() in
    Printf.printf "Le consommateur %s prend le produit\n" mag
    flush stdout ;
    Thread.delay(Random.float(3.0))
  done ;;
val consommateur :
  < prendre : unit -> < nom : string; .. >; .. > ->

```

4. Définir une fonction **creer_produit** de type *string -> produit*.
 Le nom donné à un produit sera composé de la chaîne passée en argument concaténée à un numéro de produit incrémenté à chaque appel de la fonction.
 Utiliser cette fonction pour définir **producteur** : magasin -> string -> unit.

```

# let producteur =
  let num = ref 0 in
  let creer_produit () =
    let p = new produit("lessive-"^(string_of_int !num)) in
    incr num ;
    p
  in
  function mag -> function nm ->
    while true do
      let p = creer_produit () in
      mag#deposer(p) ;
      Printf.printf"Production de %s\n" p#nom ;
    end

```

```
flush stdout ;  
Thread.delay (Random.float (1.0))  
done ;;  
val producteur : < deposer : produit -> '_a; ... >
```

Résumé

Ce chapitre a abordé le domaine de la programmation concurrente où plusieurs processus interagissent soit sur une mémoire partagée, soit par une communication synchrone. Le premier cas représente la concurrence pour la programmation impérative. En particulier, nous avons détaillé les mécanismes d'exclusion mutuelle dont la mise en oeuvre permet la synchronisation des processus pour l'accès à la mémoire partagée. La communication synchrone offre un modèle pour la concurrence en programmation fonctionnelle. En particulier, la possibilité d'envoyer des fermetures et des événements de synchronisation sur les canaux de communication autorise une composition des calculs des différents processus.

Les processus utilisés dans ce chapitre sont les processus légers du module Thread d'Objective CAML.

Pour en savoir plus

Les premiers besoins en algorithmes concurrents ont surgi de la programmation système. Pour cela le modèle impératif à mémoire partagée est le plus utilisé. Par exemple, la relation d'exclusion mutuelle et les sémaphores sont utilisés pour la gestion des ressources communes. Les différents mécanismes de bas niveau de gestion de processus accédant à une mémoire partagée sont décrits dans [[Ari86](#)].

Néanmoins, la possibilité d'exprimer des algorithmes concurrents dans son langage de prédilection permet de s'intéresser à ce type d'algorithmique, comme présenté dans l'ouvrage [[And91](#)]. Il est à noter que la conception de tels algorithmes peut simplifier la résolution de certains problèmes, mais la mise au point des programmes correspondants est assez ardue.

Le modèle de la communication synchrone présenté par CML, et repris par le module Event, est bien décrit dans [Rep92]. La version en ligne est à l'adresse suivante :

Lien: <http://cm.bell-labs.com/cm/cs/who/jhr/index.html>

Un exemple intéressant est la bibliothèque graphique de processus légers, EXene, implantée en CML au dessus de X-Windows. Le lien précédent contient un pointeur sur cette bibliothèque.

Programmation répartie

Introduction

La programmation répartie permet de construire des applications fonctionnant sur plusieurs machines reliées ensemble sur un réseau et communiquant entre elles pour accomplir une tâche. Le modèle sous-jacent est celui de la programmation parallèle à mémoire répartie. Les programmes locaux ou distants communiquent via un média qui correspond à un protocole réseau. Le plus connu et le plus utilisé d'entre eux est IP (Internet protocol) et ses surcouches TCP et UDP. À partir de ces communications de bas niveau, de nombreux services sont bâtis sur le modèle client-serveur où un serveur attend des requêtes de différents clients, les traite et envoie les réponses à ces demandes. On peut citer le protocole HTTP permettant la communication entre navigateur et serveur Web. La répartition des tâches entre clients et serveurs correspond à différentes architectures logicielles.

Le langage Objective CAML offre, via sa bibliothèque Unix, différentes possibilités de communication entre programmes. Les prises de communication (sockets en anglais) permettent la communication selon le protocole TCP/IP et UDP/IP. Cette partie de la bibliothèque Unix est portée sous Windows. La possibilité de dupliquer des processus `< >` (Unix.fork) ou de créer des processus légers (Thread.create) permet de réaliser des serveurs acceptant plusieurs requêtes en même temps. À partir de ces couches de bas niveau, il est aisé de construire des client-serveur universels paramétrés par une fonction de traitement. Enfin, un point

important pour la création de nouveaux services est la définition d'un protocole propre à l'application.

Plan du chapitre

Ce chapitre présente les éléments de base du réseau Internet que sont les prises de communication, pour la construction d'applications réparties, en particulier les applications client-serveur, en détaillant les difficultés qu'il y a à réaliser des protocoles de communication.

La première section explique brièvement le réseau Internet, son adressage et ses principaux services.

La deuxième section montre la communication, à travers des sockets, de différents processus Objective CAML, locaux ou distants.

La troisième section décrit le modèle client-serveur en commentant les programmes de serveurs et de clients universels.

La quatrième section montre l'importance des protocoles de communication pour la construction de services réseau.

Ce chapitre gagne à être lu après les chapitres sur la programmation système (chapitre 18) et sur la programmation concurrente (chapitre 19).

Réseau Internet

Internet est un réseau de réseaux (Inter Networking).

L'interconnexion est hiérarchisée en domaines, sous-domaines, etc. jusqu'aux interfaces. Une interface est l'équipement matériel d'une machine permettant sa connexion (typiquement : une carte ethernet). Certaines machines peuvent posséder plusieurs interfaces. Chaque interface possède une adresse IP unique qui respecte, en général, la hiérarchie d'interconnexion. L'acheminement, ou routage des messages peut ainsi être lui-même hiérarchisé : de domaine à domaine ; puis de domaine à sous domaines, etc. jusqu'à l'interface ultime. Outre l'adresse de leur(s) interface(s), les machines ont la plupart du temps un nom, de même que les domaines, sous-

domaines, etc.

Certaines machines ont dans ce méta réseau un rôle particulier :

les passerelles

opèrent le passage d'un réseau à un autre ;

les routeurs

connaissent, en partie, la topologie d'Internet et opèrent l'acheminement des données ;

les serveurs de noms

connaissent la correspondance entre noms de machines et adresses réseau.

L'intérêt d'Internet (i.e. : du protocole IP) est que le réseau de réseaux devient une seule entité. C'est pourquoi on peut parler **du** réseau Internet. Deux machines quelconques connectées sur Internet peuvent communiquer. Différents types de machines et de systèmes cohabitent sur Internet. Elles parlent toutes les protocoles IP et, majoritairement, ses surcouches UDP et TCP.

Les différents protocoles et services du réseau Internet sont décrits dans les RFC (Request For Comments), autrement dit < >, que l'on trouve sur le site miroir de Jussieu :

Lien: <ftp://ftp.lip6.fr/pub/rfc>

Protocoles et services Internet

L'unité de transfert du protocole IP est le datagramme ou paquet IP. C'est un protocole non fiable : il n'assure ni le bon ordre, ni le bon port, ni la non duplication des datagrammes transmis. Il traite juste le routage correct des paquets et la signalisation d'erreurs lorsqu'un datagramme n'a pu arriver à destination. Un datagramme contient un en-tête et des données. Dans l'en-tête apparaissent les adresses du destinataire et de l'expéditeur du datagramme. Les adresses sont codées sur 32 bits dans la version courante du protocole : IPv4. Ces 32 bits sont scindés en 4 champs comportant une valeur comprise entre 0 et 255. On note ces adresses sous forme de quatre entiers séparés par un point comme par exemple : 132.227.60.30.

Le protocole IP connaît à l'heure actuelle une réforme importante rendue nécessaire par l'épuisement de l'espace d'adressage et la complexité croissante des problèmes de routage dûs à l'expansion d'Internet. La nouvelle version du protocole IP est IPv6 dont on peut trouver une présentation dans [Ciz98].

Au dessus d'IP, deux protocoles permettent des transmissions de plus haut niveau : UDP (User Datagram Protocol) et TCP (Transfert Control Protocol). Ces deux protocoles qui utilisent IP pour la communication entre machines permettent en plus la communication entre des applications (ou programmes) tournant sur ces machines. Ils s'occupent de la transmission correcte de l'information indépendamment du contenu. L'identification des applications sur une machine est faite en utilisant un numéro de port.

UDP est un protocole sans connexion et non fiable, il est aux applications ce qu'IP est aux interfaces. TCP est un protocole orienté connexion et fiable : il gère l'acquittement, la retransmission et l'ordonnancement des paquets. De plus, il est capable d'optimiser la transmission par une technique de fenêtrage.

Les services standard (applications) d'Internet utilisent le plus souvent le modèle client-serveur. Le serveur gère les requêtes des clients à qui il offre un service spécifique. Il y a une asymétrie entre le client et le serveur. Ces services implantent des protocoles de plus haut niveau tenant compte cette fois du contenu de l'information transmise. Parmi les services standard, on peut citer :

- FTP (File Transfert Protocol)
- TELNET (Terminal Transfert Protocol)
- SMTP (Simple Mail Transfert Protocol)
- HTTP (Hyper Text Transfert Protocol)

D'autres services utilisent ce modèle client-serveur :

- NFS (Network File System)
- X-Windows
- les services Unix : rlogin, rwho ...

La communication entre applications s'effectue via des prises de

communication. Elles permettent la communication entre des processus ne résidant pas forcément sur une même machine. Différents processus peuvent lire et écrire dans cette voie de communication.

Module

Le module Unix fournit le type abstrait *inet_addr* représentant les adresses Internet ainsi que deux fonctions de conversion entre représentation interne des adresses et chaînes de caractères :

```
# Unix.inet_addr_of_string ;;
- : string -> Unix.inet_addr = <fun>
# Unix.string_of_inet_addr ;;
- : Unix.inet_addr -> string = <fun>
```

Dans les applications, les adresses Internet et les numéros de port de services (ou numéros de services) sont souvent remplacés par des noms. La correspondance entre noms et adresse ou numéro est gérée à partir de bases de données. Le module Unix fournit des fonctions de requête sur ces bases et les types de données permettant le stockage des informations obtenues. Nous les décrivons brièvement ci-dessous.

Base d'adresses

La base d'adresses (hosts database) contient l'association entre nom(s) de machine et adresse(s) d'interface(s). La structure des entrées de la base d'adresses est représentée par :

```
# type host_entry =
{ h_name : string;
  h_aliases : string array;
  h_addrtype : socket_domain;
  h_addr_list : inet_addr array } ;;
```

Les deux premiers champs contiennent le nom de la machine et ses

alias, le troisième, le type d'adresse (voir page ??) et le dernier, la liste des adresses des interfaces de la machine.

On obtient le nom de sa machine par la fonction :

```
# Unix.gethostname ;;
- : unit -> string = <fun>
# let my_name = Unix.gethostname() ;;
val my_name : string = "zinc.pps.jussieu.fr"
```

Les fonctions d'interrogation de la base d'adresses nécessitent en entrée soit le nom, soit l'adresse de la machine.

```
# Unix.gethostbyname ;;
- : string -> Unix.host_entry = <fun>
# Unix.gethostbyaddr ;;
- : Unix.inet_addr -> Unix.host_entry = <fun>
# let my_entree_byname = Unix.gethostbyname my_name ;;
val my_entree_byname : Unix.host_entry =
  {Unix.h_name="zinc.pps.jussieu.fr"; Unix.h_aliases=[| "zinc.pps.jussieu.fr" |];
   Unix.h_addrtype=Unix.PF_INET; Unix.h_addr_list=[| <abstr> |]}
# let my_addr = my_entree_byname.Unix.h_addr_list.(0) ;;
val my_addr : Unix.inet_addr = <abstr>

# let my_entree_byaddr = Unix.gethostbyaddr my_addr ;;
val my_entree_byaddr : Unix.host_entry =
  {Unix.h_name="zinc.pps.jussieu.fr"; Unix.h_aliases=[| "zinc.pps.jussieu.fr" |];
   Unix.h_addrtype=Unix.PF_INET; Unix.h_addr_list=[| <abstr> |]}

# let my_full_name = my_entree_byaddr.Unix.h_name ;;
val my_full_name : string = "zinc.pps.jussieu.fr"
```

Ces fonctions déclenchent l'exception `Not_found` en cas d'échec de la requête.

Base de services

La base de services contient la correspondance entre noms de

service et numéros de port. La plupart des services d'Internet sont standardisés. La structure des entrées de la base de services est :

```
# type service_entry =  
  { s_name : string;  
    s_aliases : string array;  
    s_port : int;  
    s_proto : string } ;;
```

Les premiers champs sont le nom du service et ses éventuels alias, le troisième contient le numéro de port du service et le dernier, le nom du protocole utilisé.

Un service est en fait caractérisé par son numéro de port et son protocole. Les fonctions d'interrogation sont :

```
# Unix.getservbyname ;;  
- : string -> string -> Unix.service_entry = <fun>  
# Unix.getservbyport ;;  
- : int -> string -> Unix.service_entry = <fun>  
# Unix.getservbyport 80 "tcp" ;;  
- : Unix.service_entry =  
{Unix.s_name="http"; Unix.s_aliases=[|]|]; Unix.s_port=80}  
# Unix.getservbyname "ftp" "tcp" ;;  
- : Unix.service_entry =  
{Unix.s_name="ftp"; Unix.s_aliases=[|]|]; Unix.s_port=21};
```

Ces fonctions déclenchent l'exception `Not_found` si elles ne trouvent pas le service demandé.

Prises de communication

Nous avons vus aux chapitres [18](#) et [19](#) deux moyens de communication entre processus, respectivement : les tubes et les canaux. Ces deux premières méthodes utilisent un modèle logique de concurrence. Elles n'induisent en général pas d'amélioration de performances dans la mesure où les processus communicants partagent les mêmes ressources ; en particulier, le même processeur.

La troisième possibilité, que nous présentons dans ce paragraphe, utilise les prises de communication. Cette possibilité vient du monde Unix. Elle permet la communication entre processus s'exécutant soit sur une même machine, soit sur des machines différentes.

Description et création

Une prise de communication a pour rôle d'établir une communication avec une autre prise dans le but de transférer des informations. Nous énumérons les différentes situations que l'on peut rencontrer ainsi que les commandes et types qui seront utilisés par les sockets TCP/IP. La métaphore classique est de comparer les sockets aux postes téléphoniques.

- Pour fonctionner ils doivent être raccordés au réseau (socket).
- Pour recevoir un appel ils doivent posséder un numéro de type *sock_addr* (bind).
- Pendant un appel, il est possible de recevoir un autre appel si la configuration le permet (listen).
- Il n'est pas nécessaire d'avoir soi-même un numéro pour appeler un autre poste, une fois la connexion établie la communication est dans les deux sens (connect).

Domaine

Suivant qu'une prise est destinée à la communication interne ou externe elle appartient à un domaine différent. La bibliothèque Unix définit deux domaines possibles correspondant aux constructeurs du type :

```
# type socket_domain = PF_UNIX | PF_INET;;
```

Le premier domaine correspond à la communication locale et le second, à la communication transitant sur le réseau Internet. Ce sont là les principaux domaines d'application des sockets.

Nous n'utiliserons dans la suite que les sockets du domaine Internet.

Type et protocole

Quel que soit leur domaine, les sockets définissent certaines propriétés de communication (fiabilité, ordonnancement, etc.) représentées par les constructeurs du type :

```
# type socket_type = SOCK_STREAM | SOCK_DGRAM | SOCK_SEQ
```

Suivant le type de socket utilisé, le protocole sous-jacent à la communication obéira aux caractéristiques définies. À chaque type de communication est associé un protocole par défaut.

En fait, nous n'utiliserons ici que le premier type de communication : `SOCK_STREAM` avec le protocole par défaut TCP. Il garantit fiabilité, ordonnancement et non duplication des messages échangés et fonctionne en mode connecté.

Pour le reste, nous renvoyons le lecteur à la littérature sur Unix, par exemple [\[Rif90\]](#).

Création

La fonction de création d'une socket est :

```
# Unix.socket ;;  
- : Unix.socket_domain -> Unix.socket_type -> int -> Uni
```

Le troisième argument permet de préciser le protocole associé à la communication. La valeur 0 est interprétée comme « le protocole par défaut » associé au couple (domaine, type), argument de la création de la socket. La valeur de retour de cette fonction est un descripteur de fichier. Ainsi les échanges pourront se faire en utilisant les fonctions standard du module Unix d'entrées-sorties.

Créons une socket TCP/IP :

```
# let s_descr = Unix.socket Unix.PF_INET Unix.SOCK_STREAM
```

```
val s_descr : Unix.file_descr = <abstr>
```

Warning

Bien que la fonction `socket` retourne une valeur de type *file_descr*, le système fait la différence entre un descripteur de fichier classique et celui attribué à une socket.

On peut utiliser les fonctions sur les fichiers du module `Unix` avec les sockets ; mais une exception est déclenchée dans le cas où un descripteur classique est passé à une fonction attendant une socket.

Fermeture

Comme tout descripteur de fichier, une socket se ferme par la fonction :

```
# Unix.close ;;  
- : Unix.file_descr -> unit = <fun>
```

La sortie par `exit` d'un processus ferme les descripteurs de fichier encore ouverts.

Adresses et connexions

Une socket que l'on vient de créer ne possède pas d'adresse. Pour qu'une connexion puisse s'établir entre deux sockets, l'appelant doit connaître l'adresse du récepteur.

L'adresse d'une socket (TCP/IP) est constituée d'une adresse IP et d'un numéro de port. Une socket du domaine `Unix` est simplement constituée d'un nom de fichier.

```
# type sockaddr =  
  ADDR_UNIX of string | ADDR_INET of inet_addr * int ;;
```

Attribution d'une adresse

La première chose à faire pour pouvoir recevoir des appels après la création d'une socket est de lui attribuer (de la lier à) une adresse. C'est le rôle de la fonction :

```
# Unix.bind ;;  
- : Unix.file_descr -> Unix.sockaddr -> unit = <fun>
```

En effet, nous avons déjà un descripteur de socket, mais l'adresse qui lui est attachée à la création ne peut guère nous être utile comme le montre l'exemple suivant :

```
# let (addr_in, p_num) =  
  match Unix.getsockname s_descr with  
  | Unix.ADDR_INET (a,n) -> (a,n)  
  | _ -> failwith "pas_INET";;  
val addr_in : Unix.inet_addr = <abstr>  
val p_num : int = 0  
# Unix.string_of_inet_addr addr_in;;  
- : string = "0.0.0.0"
```

Reste donc à fabriquer une adresse utile et à l'associer à notre socket. Nous reprenons notre adresse locale `my_addr` obtenue page ?? et choisissons le port 12345 qui, en général, n'est pas attribué.

```
# Unix.bind s_descr (Unix.ADDR_INET(my_addr, 12345)) ;;  
- : unit = ()
```

Capacité d'écoute et réception

Il faut encore procéder à deux opérations avant que notre socket soit complètement opérationnelle pour recevoir des appels : définir sa capacité d'écoute et se mettre effectivement à l'écoute. C'est le rôle respectif des deux fonctions :

```
# Unix.listen ;;
- : Unix.file_descr -> int -> unit = <fun>
# Unix.accept ;;
- : Unix.file_descr -> Unix.file_descr * Unix.sockaddr =
```

Le second argument de la fonction `listen` indique le nombre maximal de connexions toléré en attente. L'appel de la fonction `accept` attend une demande de connexion. Quand celle-ci survient la fonction `accept` se termine et retourne l'adresse de la socket ayant appelé ainsi qu'un nouveau descripteur de socket, dite socket de service. Cette socket de service est automatiquement liée à une adresse. La fonction `accept` ne s'applique qu'aux sockets ayant exécuté un `listen`, c'est-à-dire aux sockets ayant paramétré la file d'attente des demandes de connexion.

Demande de connexion

La fonction duale de `accept` est ;

```
# Unix.connect ;;
- : Unix.file_descr -> Unix.sockaddr -> unit = <fun>
```

Un appel à `Unix.connect s_descr s_addr` établit une connexion entre la socket locale `s_descr` (qui est automatiquement liée) et la socket d'adresse `s_addr` (qui doit exister).

Communication

À partir du moment où une connexion est établie entre deux sockets, les processus propriétaires de ces dernières peuvent communiquer dans les deux sens. Les fonctions d'entrées-sorties sont celles du module `Unix` décrites au chapitre [18](#).

Client-serveur

La communication entre processus sur une même machine ou sur

des machines différentes à travers des sockets TCP/IP est un mode de communication point-à-point asynchrone. La fiabilité des transmissions est assurée par le protocole TCP. Il est néanmoins possible de simuler une diffusion à un ensemble de processus en effectuant des communications point-à-point sur tous les récepteurs.

Le rôle des différents processus entrant en jeu dans la communication d'une application est en règle générale asymétrique. C'est le cas pour les architectures client-serveur. Un serveur est un processus (ou plusieurs) acceptant des requêtes et tâchant d'y répondre. Le client, lui-même un processus, envoie une requête au serveur en espérant une réponse.

Schéma d'actions d'un client-serveur

Un serveur ouvrira un service sur un port donné et se mettra en attente de connexions de la part de futurs clients. La figure 20.1 montre le déroulement des principales tâches d'un serveur et d'un client.

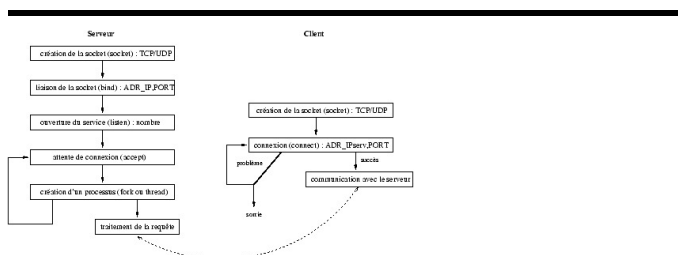


Figure 20.1 : schéma des actions d'un serveur et d'un client

Un client peut se connecter à un service à partir du moment où le serveur est à la phase d'acceptation de connexions (accept). Pour cela il devra connaître le numéro IP de la machine serveur et le numéro de port du service. S'il ne connaît pas le numéro IP, il devra demander une résolution nom/numéro par la fonction

gethostbyname. Une fois la connexion acceptée par le serveur, chaque programme pourra communiquer via les canaux d'entrées-sorties de la socket créée de part et d'autre.

Programmation d'un client-serveur

La mécanique de programmation d'un client-serveur va donc suivre le schéma décrit à la figure 20.1. Ces tâches sont à réaliser dans tous les cas. Pour cela nous écrirons les fonctions génériques d'agencement de ces tâches, paramétrées par les fonctions particulières à un serveur donné. On illustrera ces programmes par un premier serveur qui accepte une connexion à partir d'un client, attend sur cette prise de communication qu'une ligne soit passée, la convertit en MAJUSCULES et la renvoie convertie au client.

La figure 20.2 montre la communication entre ce service et différents clients.

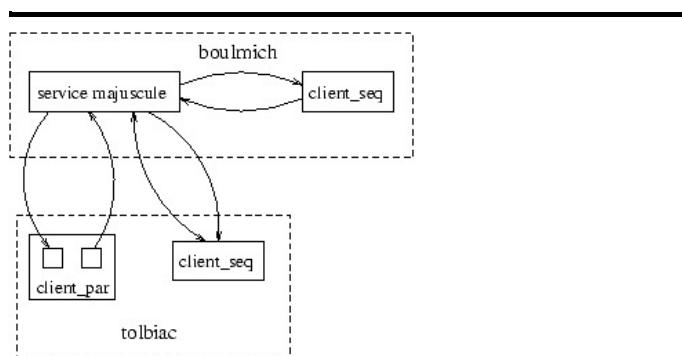


Figure 20.2 : service MAJUSCULE et des clients

Certains tournent sur la même machine que le serveur et d'autres se trouvent sur des machines distantes.

Dans la suite de ce paragraphe nous verrons

1. Comment écrire le code d'un « serveur générique » pour

- l'instancier en notre service particulier de mise en majuscules.
2. Comment tester ce serveur, sans avoir à écrire de client, en utilisant la commande telnet.
 3. Comment implanter deux types de clients :
 - le premier, séquentiel, c'est à dire qu'après l'envoi d'une requête, il attend la réponse ;
 - le second, parallèle, qui sépare les tâches d'envoi et de réception. Il y aura donc deux processus pour ce client.

Code du serveur

Un serveur se découpe en deux parties : l'attente d'une connexion et le traitement suite à une connexion.

Serveur générique

Le serveur générique `establish_server` décrit ci-dessous est une fonction qui prend en premier argument la fonction de service (`server_fun`) chargée de traiter les requêtes et, en second, l'adresse de la socket dans le domaine Internet qui sera à l'écoute des requêtes. Cette fonction utilise la fonction auxiliaire `domain_of` qui extrait le domaine d'une socket à partir de son adresse.

En fait, la fonction `establish_server` fait partie des fonctions de haut niveau du module Unix. Nous en donnons l'implantation de la distribution.

```
# let establish_server server_fun sockaddr =  
  let domain = domain_of sockaddr in  
  let sock = Unix.socket domain Unix.SOCK_STREAM 0  
  in Unix.bind sock sockaddr ;  
  Unix.listen sock 3;  
  while true do  
    let (s, caller) = Unix.accept sock  
    in match Unix.fork() with
```

```

0 -> if Unix.fork() <> 0 then exit 0 ;
let inchan = Unix.in_channel_of_descr s
and outchan = Unix.out_channel_of_descr s
in server_fun inchan outchan ;
close_in inchan ;
close_out outchan ;
exit 0
| id -> Unix.close s; ignore(Unix.waitpid [] id)
done ;;

val establish_server :
  (in_channel -> out_channel -> 'a) -> Unix.sockaddr -> unit

```

Pour construire complètement un serveur en tant qu'exécutable autonome paramétré par le numéro de port, on écrit la fonction `main_serveur` qui prend toujours en paramètre la fonction de service. Elle utilise le paramètre de la ligne de commande comme numéro de port du service. On utilise la fonction auxiliaire `get_my_addr` qui retourne l'adresse de la machine locale.

```

# let get_my_addr () =
  (Unix.gethostname(Unix.gethostname())) . Unix.h_addr_list
val get_my_addr : unit -> Unix.inet_addr = <fun>

# let main_serveur serv_fun =
  if Array.length Sys.argv < 2 then Printf.eprintf "usage"
  else try
    let port = int_of_string Sys.argv.(1) in
    let mon_adresse = get_my_addr()
    in establish_server serv_fun (Unix.ADDR_INET(mon_adresse))
  with
    Failure("int_of_string") ->
      Printf.eprintf "serv_up : bad port number\n" ;;
val main_serveur : (in_channel -> out_channel -> 'a) -> unit

```

Code du service

La mécanique générale est en place. Pour l'illustrer, il reste à définir le service. Celui-ci est un convertisseur de chaînes en majuscules. Il

attend une ligne sur le canal d'entrée, la convertit et l'écrit sur le canal de sortie en vidant le tampon.

```
# let uppercase_service ic oc =  
  try while true do  
    let s = input_line ic in  
    let r = String.uppercase s  
    in output_string oc (r^"\n") ; flush oc  
  done  
  with _ -> Printf.printf "Fin du traitement\n" ; flush s  
val uppercase_service : in_channel -> out_channel -> unit
```

Pour récupérer correctement les exceptions provenant du module Unix, on encapsule l'appel au démarrage du service dans la fonction *ad hoc* du module Unix :

```
# let go_uppercase_service () =  
  Unix.handle_unix_error main_serveur uppercase_service ;  
val go_uppercase_service : unit -> unit = <fun>
```

Compilation et test du service

On regroupe ces fonctions dans le fichier `serv_up.ml` auquel on ajoute un appel effectif à la fonction `go_uppercase_service`. On compile ce fichier en précisant l'utilisation du module Unix.

```
ocamlc -i -custom -o serv_up.exe unix.cma serv_up.ml -co
```

L'affichage de la compilation (option `-i`) donne :

```
val establish_server :  
  (in_channel -> out_channel -> 'a) -> Unix.sockaddr ->  
val main_serveur : (in_channel -> out_channel -> 'a) ->  
val uppercase_service : in_channel -> out_channel -> unit  
val go_uppercase_service : unit -> unit
```

On lance le serveur en écrivant :

```
serv_up.exe 1400
```

Le port choisi est ici 1400. Maintenant la machine où a été lancée

cette commande accepte les connexions sur ce port.

Tester avec

On peut d'ores et déjà tester le serveur en utilisant un client existant d'envoi et de réception de lignes de caractères. L'utilitaire telnet, qui normalement est un client du service telnetd sur le port 23 et que l'on utilise alors comme commande de connexion distante peut être détourné de son rôle si on lui passe en argument une machine et un autre numéro de port. Cet utilitaire existe sur les différents systèmes d'exploitation. Pour tester notre serveur, sous Unix, on tapera :

```
$ telnet boulmich 1400
Trying 132.227.89.6...
Connected to boulmich.ufr-info-p6.jussieu.fr.
Escape character is '^]'.
```

L'adresse IP de boulmich est 132.227.89.6 et son nom complet, qui contient son nom de domaine, est boulmich.ufr-info-p6.jussieu.fr. C'est bien ce qu'affiche telnet. Le client attend une frappe au clavier et l'envoie au serveur que nous avons lancé sur boulmich avec le port 1400. Il attendra la réponse du serveur et l'affichera :

```
Le petit chat est mort.
LE PETIT CHAT EST MORT.
On obtient bien le résultat escompté.
ON OBTIENT BIEN LE RÉSULTAT ESCOMPTÉ.
```

Les phrases entrées par l'utilisateur sont en minuscules et celles renvoyées par le serveur sont en majuscules. C'est justement le rôle de ce mini-service que d'assurer cette conversion.

Pour sortir de ce client il sera nécessaire soit de fermer la fenêtre d'où il a été exécuté, soit d'utiliser la commande kill. La socket de communication du client sera alors fermée, ce qui provoquera du côté serveur la disparition de la socket de service. À ce moment là le serveur affiche le message « Fin de traitement », puis le processus associé à la fonction de service termine.

Code du client

Autant le serveur est naturellement parallèle (on désire traiter une requête tout en en acceptant d'autres, jusqu'à une certaine limite), autant le client peut l'être ou ne pas l'être selon la nature de l'application à développer. Nous donnerons ci-dessous deux versions de client. Mais auparavant, nous présentons deux fonctions utiles pour l'écriture de ces clients.

La fonction `open_connection` du module `Unix` permet à partir d'une socket INET d'obtenir un couple de canaux classiques d'entrées-sorties sur cette socket.

Le code suivant est issu de la distribution du langage.

```
# let open_connection sockaddr =  
  let domain = domain_of sockaddr in  
  let sock = Unix.socket domain Unix.SOCK_STREAM 0  
  in try Unix.connect sock sockaddr ;  
    (Unix.in_channel_of_descr sock , Unix.out_channel_of_descr sock)  
  with exn -> Unix.close sock ; raise exn ;;  
val open_connection : Unix.sockaddr -> in_channel * out_channel
```

De même, la fonction `shutdown_connection` effectue la fermeture en envoi de la socket.

```
# let shutdown_connection inchan =  
  Unix.shutdown (Unix.descr_of_in_channel inchan) Unix.SHUT_RDWR  
val shutdown_connection : in_channel -> unit = <fun>
```

Client séquentiel

À partir de ces fonctions on peut écrire la fonction principale du client prenant en argument la fonction d'envoi de requêtes et de réception des réponses. Elle analyse les arguments de la liste de commande pour obtenir les paramètres de connexion avant de

lancer le traitement.

```
# let main_client client_fun =
  if Array.length Sys.argv < 3
  then Printf.printf "usage :  client serveur port\n"
  else let serveur = Sys.argv.(1) in
  let serveur_adr =
  try Unix.inet_addr_of_string serveur
  with Failure("inet_addr_of_string") ->
  try (Unix.gethostbyname serveur).Unix.h_addr_list.(0)
  with Not_found ->
  Printf.eprintf "%s : serveur inconnu\n" serveur ;
  exit 2
  in try
  let port = int_of_string (Sys.argv.(2)) in
  let sockadr = Unix.ADDR_INET(serveur_adr,port) in
  let ic,oc = open_connection sockadr
  in client_fun ic oc ;
  shutdown_connection ic
  with Failure("int_of_string") -> Printf.eprintf "bad port\n" ;
  exit 2 ;;
val main_client : (in_channel -> out_channel -> 'a) -> unit
```

Il ne reste plus qu'à écrire la fonction de traitement du client.

```
# let client_fun ic oc =
  try
  while true do
  print_string "Requête : " ;
  flush stdout ;
  output_string oc ((input_line stdin) ^ "\n") ;
  flush oc ;
  let r = input_line ic
  in Printf.printf "Réponse : %s\n\n" r;
  if r = "FIN" then ( shutdown_connection ic ; raise Exit)
  done
  with
  Exit -> exit 0
  | exn -> shutdown_connection ic ; raise exn ;;
val client_fun : in_channel -> out_channel -> unit = <fu>
```

La fonction `client_fun` entre dans une boucle a priori sans fin qui lit le clavier, envoie la chaîne au serveur, récupère la chaîne transformée en majuscules et l'affiche. Si la chaîne vaut "FIN" l'exception `Exit` est déclenchée pour sortir de la boucle. Si une autre exception est déclenchée, typiquement si le serveur disparaît, la fonction interrompt son calcul.

Le programme client devient donc :

```
# let go_client () = main_client client_fun ;;  
val go_client : unit -> unit = <fun>
```

On regroupe toutes ces fonctions dans un fichier nommé `client_seq.ml` en ajoutant l'appel à la fonction `go_client`. On le compile ensuite avec la ligne de commande suivante :

```
ocamlc -i -custom -o client_seq.exe unix.cma client_seq.
```

L'exécution du client est alors la suivante :

```
$ client_seq.exe boulmich 1400  
Requête : Le petit chat est mort.  
Réponse : LE PETIT CHAT EST MORT.
```

```
Requête : On obtient le résultat escompté.  
Réponse : ON OBTIENT LE RÉSULTAT ESCOMPTÉ.
```

```
Requête : fin  
Réponse : FIN
```

Client parallèle avec fork

Le client parallèle proposé ici répartit sa tâche sur deux processus : l'un d'émission et l'autre de réception. Ils partagent la même socket. Les fonctions associées à chacun des processus sont passées en paramètre.

Voici le texte du programme ainsi modifié.

```

# let main_client client_pere_fun client_fils_fun =
  if Array.length Sys.argv < 3
  then Printf.printf "usage :  client serveur port\n"
  else
    let serveur = Sys.argv.(1) in
    let serveur_adr =
      try Unix.inet_addr_of_string serveur
      with Failure("inet_addr_of_string")
      -> try (Unix.gethostbyname serveur).Unix.h_addr_list.(0)
      with Not_found ->
        Printf.eprintf "%s : serveur inconnu\n" serveur ;
        exit 2
    in try
      let port = int_of_string (Sys.argv.(2)) in
      let sockadr = Unix.ADDR_INET(serveur_adr,port) in
      let ic,oc = open_connection sockadr
      in match Unix.fork () with
      0 -> if Unix.fork() = 0 then client_fils_fun oc ;
        exit 0
        | id -> client_pere_fun ic ;
        shutdown_connection ic ;
        ignore (Unix.waitpid [] id)
      with
        Failure("int_of_string") -> Printf.eprintf "bad port number\n" ;
        exit 2 ;;
val main_client : (in_channel -> 'a) -> (out_channel -> 'b) -> unit

```

Le comportement attendu des paramètres est : le (petit-)fils envoie la requête et le père reçoit la réponse.

Cette architecture prend du sens si le fils doit envoyer plusieurs requêtes, le père recevra les réponses des premières requêtes au fur et à mesure de leur traitement. On reprend donc l'exemple précédent de conversion de chaînes en majuscules mais en modifiant le côté client. Celui-ci lit le texte à convertir dans un fichier et écrit la réponse dans un autre fichier. Pour cela nous aurons besoin d'une fonction de copie d'un canal (ic) dans un autre (oc) respectant notre petit protocole (c'est à dire reconnaissant la chaîne "FIN").

```
# let copie_canaux ic oc =
  try while true do
    let s = input_line ic
    in if s = "FIN" then raise End_of_file
    else (output_string oc (s^"\n"); flush oc)
  done
  with End_of_file -> () ;;
val copie_canaux : in_channel -> out_channel -> unit = <
```

On écrit les deux fonctions destinées au fils et au père du schéma de client parallèle :

```
# let fils_fun in_file out_sock =
  copie_canaux in_file out_sock ;
  output_string out_sock ("FIN\n") ;
  flush out_sock ;;
val fils_fun : in_channel -> out_channel -> unit = <fun>
# let pere_fun out_file in_sock = copie_canaux in_sock out_file
val pere_fun : out_channel -> in_channel -> unit = <fun>
```

Cela permet d'écrire la fonction principale du client. Elle devra récupérer sur la ligne de commande deux paramètres supplémentaires : le nom du fichier d'entrée et le nom du fichier de sortie.

```
# let go_client () =
  if Array.length Sys.argv < 5
  then Printf.eprintf "usage : client_par serveur port fichier"
  else let in_file = open_in Sys.argv.(3)
    and out_file = open_out Sys.argv.(4)
    in main_client (pere_fun out_file) (fils_fun in_file) ;
    close_in in_file ;
    close_out out_file ;;
val go_client : unit -> unit = <fun>
```

On réunit tout notre matériel dans le fichier client_par.ml (sans oublier l'appel effectif à go_client), on compile. On crée alors le fichier toto.txt contenant le texte à convertir, disons :

Le petit chat est mort.
On obtient le résultat escompté.

On peut alors tester en tapant :

```
client_par.exe boulmich 1400 toto.txt result.txt
```

Le fichier result.txt contient le texte :

```
$ more result.txt  
LE PETIT CHAT EST MORT.  
ON OBTIENT LE RÉSULTAT ESCOMPTÉ.
```

Lorsque le client termine, le serveur affiche toujours le message "Fin de traitement".

Client-serveur avec processus légers

La présentation précédente du code d'un serveur générique et d'un client parallèle utilise la création de nouveaux processus grâce à la primitive fork du module Unix. Cela fonctionne bien sous Unix et de nombreux services Unix sont mis en oeuvre par cette technique. Ce n'est cependant pas le cas avec Windows. Pour la portabilité, on écrira de préférence les client-serveur avec les processus légers qui ont été présentés au chapitre 19. Il sera nécessaire de déterminer les interactions entre les différents processus du serveur.

Threads et bibliothèque Unix

L'utilisation conjointe de la bibliothèque de processus légers et de la bibliothèque Unix provoque le blocage de tous les threads actifs si un appel système ne répond pas immédiatement. En particulier, les lectures sur un descripteur de fichiers incluant donc ceux créés par socket, sont bloquantes.

Pour éviter ce désagrément, le module ThreadUnix réimplante la plupart des fonctions d'entrées-sorties du module Unix. Les fonctions définies dans ce module ne bloqueront que le thread qui effectue l'appel système. En conséquence, les entrées-sorties devront être implantées avec les fonctions de plus bas niveau read et write offertes par le module ThreadUnix. Par exemple, on redéfinit la fonction standard de lecture d'une chaîne de caractères, input_line,

de façon à ce qu'elle ne bloque pas les autres threads pendant la lecture d'une ligne.

```
# let my_input_line fd =  
  let s = " " and r = ref ""  
  in while (ThreadUnix.read fd s 0 1 > 0) && s.[0] <> '\r'  
    !r ;;  
val my_input_line : Unix.file_descr -> string = <fun>
```

Classes pour un serveur avec threads

Nous reprenons l'exemple du service MAJUSCULE pour en donner une version utilisant les processus légers. Le passage aux threads ne pose pas de problème puisque notre petite application, aussi bien côté serveur que côté client, lance des processus fonctionnant indépendamment.

Nous avons précédemment implanté un serveur générique paramétré par une fonction de service. Nous avons réalisé cette abstraction en utilisant le caractère fonctionnel du langage Objective CAML. Nous proposons d'utiliser l'extension objet du langage pour illustrer comment les objets permettent de réaliser une abstraction analogue.

L'organisation du serveur repose sur deux classes : `serv_socket` et `connexion`. La première correspond à la mise en route du service, la seconde, à la fonction de service. Nous avons introduit quelques impressions traçant les principales étapes du service.

La classe `serv_socket`

possède deux variables d'instance : `port` et `socket` correspondant au numéro de port du service et à la socket d'écoute. À la construction d'un tel objet l'initialisateur effectue les opérations d'ouverture de service et crée cette socket. La méthode `run` se met en acceptation de connexions, et crée un nouvel objet `connexion` pour lancer le traitement de la requête. La classe `serv_socket` utilise la classe

connexion présentée au paragraphe suivant. Cette dernière doit normalement être définie avant la classe *serv_socket*.

```
# class serv_socket p =
  object (self)
    val port = p
    val mutable sock = ThreadUnix.socket Unix.PF_INET Unix.

  initializer
    let mon_adresse = get_my_addr ()
    in Unix.bind sock (Unix.ADDR_INET(mon_adresse,port)) ;
    Unix.listen sock 3

  method private client_addr = function
    Unix.ADDR_INET(host,_) -> Unix.string_of_inet_addr host
    | _ -> "Unexpected client"

  method run () =
    while(true) do
      let (sd,sa) = ThreadUnix.accept sock in
      let connexion = new connexion(sd,sa)
      in Printf.printf "TRACE.serv: nouvelle connexion de %s\n"
        (self#client_addr sa) ;
      ignore (connexion#start ())
    done
  end ;;

class serv_socket :
  int ->
  object
  val port : int
  val mutable sock : Unix.file_descr
  method private client_addr : Unix.sockaddr -> string
  method run : unit -> unit
end
```

Il est toujours possible d'affiner le serveur en héritant de cette classe et en redéfinissant la méthode *run*.

La classe connexion

Les variables d'instance de cette classe, `s_descr` et `s_addr`, seront initialisées avec le descripteur et l'adresse de la socket de service créés par `accept`. Les méthodes sont `start`, `run` et `stop`. La méthode `start` crée un thread appelant les deux autres et retourne son identificateur qui pourrait être manipulé par l'instance appelante de `serv_socket`. C'est dans la méthode `run` que l'on retrouve le corps de la fonction de service. Nous avons un peu modifié la condition de fin de service : on sort sur une chaîne vide. La méthode `stop` se contente de fermer le descripteur de la socket de service.

À chaque nouvelle connexion sera attribué un numéro obtenu par appel à la fonction auxiliaire `gen_num` lors de la création d'une instance.

```
# let gen_num = let c = ref 0 in (fun () -> incr c; !c)
val gen_num : unit -> int = <fun>
# exception Fin ;;
exception Fin
# class connexion (sd,sa) =
  object (self)
    val s_descr = sd
    val s_addr = sa
    val mutable numero = 0
  initializer
    numero <- gen_num();
    Printf.printf "TRACE.connexion : objet traitant %d créé\n" numero;
    print_newline()

  method start () = Thread.create (fun x -> self#run x ;

  method stop() =
    Printf.printf "TRACE.connexion : fin objet traitant %d\n" numero;
    print_newline () ;
    Unix.close s_descr

  method run () =
    try
      while true do
        let ligne = my_input_line s_descr
        in if (ligne = "") or (ligne = "\013") then raise Fin ;
```

```

let result = (String.uppercase ligne) ^ "\n"
in ignore (ThreadUnix.write s_descr result 0 (String.length result))
done
with
  Fin -> ()
  | exn -> print_string (Printexc.to_string exn) ; print_endline exn
end ;;

class connexion :
  Unix.file_descr * 'a ->
  object
    val mutable numero : int
    val s_addr : 'a
    val s_descr : Unix.file_descr
    method run : unit -> unit
    method start : unit -> Thread.t
    method stop : unit -> unit
  end

```

Ici encore, par héritage et redéfinition de la méthode `run`, on peut définir un nouveau service.

On testera cette nouvelle version du serveur en exécutant la fonction `protect_serv`.

```

# let go_serv () = let s = new serv_socket 1400 in s#run
# let protect_serv () = Unix.handle_unix_error go_serv ()

```

Client-serveur à plusieurs niveaux

Bien que la relation client-serveur soit asymétrique, rien n'empêche un serveur d'être lui-même client d'un autre service. On obtient ainsi une hiérarchie dans la communication. Une application client-serveur classique comporte bien souvent :

- un poste client muni d'une interface conviviale;
- un programme de traitement suite à une interaction de l'utilisateur;
- une base de données accessible par le programme de

traitement.

Un des buts des applications client-serveur est de décharger les machines centrales d'une partie du traitement. La figure 20.3 montre deux architectures client-serveur à 3 niveaux (tiers).

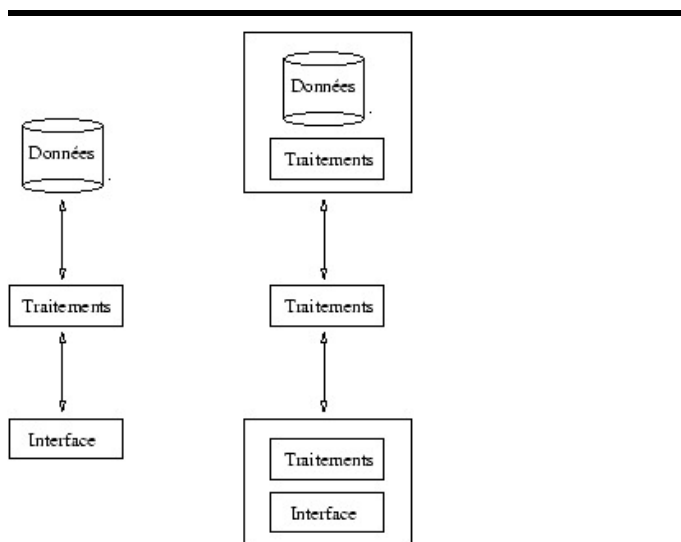


Figure 20.3 : différentes architectures de client-serveur

Chaque niveau peut être implanté sur des machines différentes. L'interface utilisateur s'exécute sur la machine d'un utilisateur de l'application. La partie traitement est localisée sur une machine commune à un ensemble d'utilisateurs qui elle-même envoie des requêtes à un serveur de base de données distant. Selon les caractéristiques de l'application, une partie des traitements peut être déportée soit sur le poste utilisateur, soit sur le serveur de base de données.

Remarques sur les client-serveur réalisés

Nous avons, dans les sections précédentes, élaboré un service simple : le service MAJUSCULE. Différentes solutions ont été exposées. Tout d'abord le serveur a utilisé le mécanisme Unix des forks. Une fois ce premier serveur construit, il a été possible de le tester par le client telnet existant sous tous les systèmes (Unix, Windows et MacOS). Ensuite un premier client simple a été écrit. Nous avons pu alors tester l'ensemble de l'application client-serveur. Les clients eux-aussi peuvent avoir des tâches à gérer entre les communications. Pour cela le client `client_par.exe`, qui sépare la lecture de l'écriture en utilisant aussi des forks, a été construit. Une nouvelle mouture du serveur a été réalisée en utilisant des threads pour bien montrer l'indépendance relative entre le serveur et le client, et l'attention à apporter aux entrées-sorties dans ce cadre. Ce serveur a été organisé sous forme de deux classes facilement réutilisables. On note que la programmation fonctionnelle et la programmation objet permettent de séparer la partie « mécanique » réutilisable de la partie traitement spécialisé.

Protocoles de communication

Les différentes communications des client-serveur décrits à la section précédente consistent à envoyer une chaîne de caractères terminée par un retour-chariot et à en recevoir une autre. Bien que simples elles définissent un protocole. Dès que l'on veut communiquer des valeurs plus complexes : un flottant, une matrice de flottants, un arbre d'expressions arithmétiques, une fermeture ou un objet, se pose le problème du codage de ces valeurs. Plusieurs solutions existent selon la nature des programmes communicants que l'on caractérise par : le langage d'implantation, l'architecture machine et dans certains cas le système d'exploitation. En effet selon l'architecture machine les entiers seront représentés de différentes manières (bits de poids fort à gauche, à droite, bits de tag, taille du mot machine). Pour communiquer une valeur entière cohérente entre différents programmes, il est nécessaire d'avoir une représentation commune des valeurs que l'on appelle représentation externe¹. Les valeurs plus structurées, comme les enregistrements, doivent également avoir une représentation externe. On rencontre néanmoins un problème lorsque certains langages autorisent des constructions, comme les champs de bits en C, qui n'existent pas

dans les autres langages. Le passage de valeurs fonctionnelles ou d'objets, qui contiennent une partie code, pose une nouvelle difficulté pour le chargement de ce code. Celui-ci est-il compatible (code-octet) entre l'expéditeur et le récepteur, et existe-t-il un mécanisme de chargement dynamique de code? En règle générale on simplifiera ce problème en supposant que le code existe des deux côtés. Ce n'est pas lui qui est transmis, mais des informations permettant de le retrouver. Pour un objet on communique ses variables d'instance et son type ce qui permet d'en retrouver les méthodes. Pour une fermeture on envoie son environnement et l'adresse de son code. Ce qui implique que les deux programmes communicants sont le même exécutable.

Une deuxième difficulté provient de la complexité des échanges liée à la nécessité de synchroniser les communications mettant en jeu plusieurs programmes.

On présente tout d'abord des protocoles texte, pour ensuite discuter de l'acquittement et des limites de temps entre requêtes et réponses. Nous évoquons aussi la difficulté de communiquer des valeurs internes d'un programme en particulier dans le cadre de l'interopérabilité entre programmes issus de différents langages.

Protocole texte

Les protocoles texte, c'est à dire la communication en format ASCII, sont les plus courants car les plus simples à mettre en oeuvre et les plus portables. Par contre quand le protocole se complique, il peut être lourd à implanter. Dans ce cadre là on définit une grammaire pour décrire le format de communication. Celui-ci peut être riche, mais ce sera aux programmes communicants de faire le travail de codage et d'interprétation des chaînes reçues.

En règle générale, l'utilisation d'une application réseau ne permet pas de voir les différentes couches de protocoles utilisées. C'est typiquement le cas du protocole HTTP permettant à un navigateur de communiquer avec un site Web.

Protocole HTTP

On rencontre fréquemment le terme < > dans les publicités. Il correspond au protocole de communication des applications du Web. Celui-ci est complètement décrit sur la page du consortium W3 :

Lien: <http://www.w3.org>

Ce protocole est utilisé pour transporter les requêtes émises par les navigateurs (Communicator, Explorer, Opera, etc) et pour retourner le contenu de la page demandée. Une requête effectuée par un navigateur contient le nom du protocole (HTTP), le nom de la machine (www.ufr-info-p6.jussieu.fr), et l'arborescence de la page demandée (/Public/Localisation/index.html). Cet ensemble de trois composants permet de définir une URL (Uniform Ressource Locators) :

`http://www.ufr-info-p6.jussieu.fr/Public/Localisation/in`

Quand une telle URL est demandée à partir d'un navigateur, une connexion via une socket est établie entre le navigateur et le serveur s'exécutant sur la machine indiquée, par défaut sur le port 80. Puis le navigateur envoie une requête au format HTTP, proche de la suivante :

```
GET /index.html HTTP/1.0
```

Alors le serveur lui répond dans le protocole HTTP, l'en-tête :

```
HTTP/1.1 200 OK
Date: Wed, 14 Jul 1999 22:07:48 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 AuthMySQL/2.20
Last-Modified: Thu, 10 Jun 1999 12:53:46 GMT
ETag: "4c801-e4f-375fb55a"
Accept-Ranges: bytes
Content-Length: 3663
Connection: close
Content-Type: text/html
```

Cet en-tête indique que la requête est acceptée (code 200 OK), la nature du serveur, la date de modification de la page, la longueur de la page envoyée et le type du contenu qui va suivre. Sans la

précision de la version de protocole (HTTP/1.0) de la commande GET, seule la page HTML est transférée. La connexion suivante avec telnet permet de voir ce qui est effectivement transmis :

```
$ telnet www.ufr-info-p6.jussieu.fr 80
Trying 132.227.68.44...
Connected to triton.ufr-info-p6.jussieu.fr.
Escape character is '^]'.
GET
```

```
<!-- index.html -->
<HTML>
<HEAD>
<TITLE>Serveur de l'UFR d'Informatique de Pierre et Mari
</HEAD>
<BODY>

<IMG SRC="/Icons/upmc.gif" ALT="logo-P6" ALIGN=LEFT HSPA
Unité de Formation et de Recherche 922 - Informatique<BR>
Université Pierre et Marie Curie<BR>
4, place Jussieu<BR>
75252 PARIS Cedex 05, France<BR><P>
....
</BODY>
</HTML>
<!-- index.html -->
```

Connection closed by foreign host.

La connexion se referme dès que la page est copiée. Le protocole de base est en mode texte ainsi que le langage à interpréter. On note que les images ne sont pas transmises avec la page. C'est au navigateur, lors de l'analyse syntaxique de la page HTML, de remarquer les ancres et les images (voir la balise IMG de la page transmise). À ce moment là, le navigateur envoie une nouvelle requête pour chaque image rencontrée dans le source HTML, il y a donc une nouvelle connexion par image. Ces images sont affichées dès qu'elles sont reçues. C'est pour cette raison que les images s'affichent souvent en parallèle.

Le protocole HTTP est assez simple, mais il véhicule une information dans le langage HTML qui lui est plus complexe.

Protocoles avec acquittement et limite de temps

Quand le protocole est complexe, il est utile que le récepteur d'un message indique à l'expéditeur qu'il a bien reçu le message et que celui-ci est grammaticalement correct. En effet le client peut être en attente bloquante de réponse avant de poursuivre ses tâches. Si la partie du serveur traitant cette requête a une difficulté d'interprétation du message, le serveur doit l'indiquer au client plutôt que d'oublier cette requête. L'exemple du protocole HTTP a une gestion de code d'erreurs. Une requête correcte renvoie le code 200. Une requête mal formée ou la demande d'accès à une page non autorisée renvoie un code d'erreur 4xx ou 5xx selon la nature de l'erreur. Ces codes d'erreur permettent au client de savoir ce qu'il a à faire et au serveur de tenir des fichiers de compte-rendu précis sur la nature des incidents. Lorsqu'un serveur est dans un état incohérent, il peut toujours accepter une connexion d'un client, mais risque de ne jamais lui envoyer de réponse sur la socket. Pour éviter ces attentes bloquantes, il est utile de fixer une limite de temps pour la transmission de la réponse. Passé ce délai, le client suppose que le serveur ne répondra plus. Alors le client peut fermer cette connexion pour passer à la suite de son travail. C'est ainsi que les navigateurs WWW font. Quand une requête n'a pas de réponse au bout d'un certain temps, le navigateur décide de l'indiquer à l'utilisateur. Objective CAML possède des entrées-sorties avec limite de temps. Dans le module Thread, les fonctions `wait_time_read` et `wait_time_write` suspendent l'exécution de la tâche jusqu'au moment où un caractère peut être lu ou écrit, cela dans un certain délai. Elles prennent en entrée un descripteur de fichier et un délai exprimé en secondes : `Unix.file_descr -> float -> bool`. Si le délai a été dépassé la fonction retourne false, sinon l'entrée-sortie peut être effectuée.

Transmission de valeurs en représentation interne

L'intérêt de la transmission de valeurs en représentation interne est

de simplifier le protocole. Il n'y a plus besoin de coder ni de décoder les données dans un format texte. Les difficultés inhérentes à l'envoi-réception de valeurs en représentation interne sont les mêmes que celles rencontrées pour les valeurs persistantes (voir le module Marshal page ??). En effet lire ou écrire une valeur dans un fichier est équivalent à recevoir ou émettre cette même valeur sur une socket.

Valeurs fonctionnelles

Dans le cas de transmission d'une fermeture entre deux programmes Objective CAML, le code de la fermeture n'est pas envoyé, seul son environnement et son pointeur de code (voir figure 12.9 page ??) le sont. Pour que cela fonctionne il est nécessaire que le serveur possède le même code au même emplacement. Ce qui implique que le même programme tourne en tant que serveur et en tant que client. Rien n'empêche cependant que ces deux programmes soient en train d'exécuter des parties de code différentes. On adapte alors le service de calcul matriciel en envoyant la fermeture contenant dans son environnement les données à calculer. À la réception le serveur applique cette fermeture à () et le calcul se déclenche.

Interopérer avec d'autres langages

L'intérêt des protocoles texte est qu'ils sont indépendants des langages d'implantation des clients et des serveurs. En effet le code ASCII est toujours connu des langages de programmation. Ensuite c'est au client et au serveur de savoir analyser syntaxiquement les chaînes de caractères transmises. Un exemple d'un tel protocole ouvert est la simulation de joueurs de football, appelée RoboCup.

Robots footballeurs

Une équipe de football joue contre une autre équipe. Chaque membre d'une équipe est un client du serveur arbitre. Les joueurs d'une même équipe ne peuvent pas communiquer directement entre eux. Ils doivent passer par le serveur, qui retransmet le dialogue. Le

serveur indique selon la position du joueur une partie du terrain. Toutes ces communications s'effectuent selon un protocole texte. La page de présentation du protocole, du serveur et de certains clients :

Lien: <http://www.robocup.org/>

Le serveur est écrit en C. Les clients sont écrits dans différents langages : C, C + + , SmallTalk, Objective CAML, etc. Rien n'empêche d'ailleurs une équipe de posséder des joueurs issus de différents langages.

Ce protocole répond aux besoins d'interopérabilité entre programmes de différents langages d'implantation. Il est relativement simple, mais il nécessite un analyseur syntaxique particulier pour chaque famille de langages.

Exercices

Les exercices proposés permettent de mettre en oeuvre différents types d'applications réparties. Le premier offre un nouveau service réseau permettant une mise à l'heure des machines clientes. Le deuxième exercice montre comment on peut utiliser les ressources de différentes machines pour distribuer un calcul.

Service : horloge

Cet exercice consiste à implanter un service « horloge » qui donne l'heure à tout client qui le désire. L'idée est d'avoir ainsi une machine de référence pour mettre à l'heure les différentes machines d'un réseau.

1. Définir un [protocole](#) pour la transmission d'une date contenant la description du jour, mois, heure, minute, seconde.

```
# type horloge = { jour:int; mois:int; heure:int; m
type horloge =
{ jour: int;
```

```

mois: int;
heure: int;
minute: int;
seconde: int }
# let encode date =
let str = String.create 5 in
str.[0] <- char_of_int date.jour ;
str.[1] <- char_of_int date.mois ;
str.[2] <- char_of_int date.heure ;
str.[3] <- char_of_int date.minute ;
str.[4] <- char_of_int date.seconde ;
str ;;
val encode : horloge -> string = <fun>
# let decode str =
{ jour = int_of_char str.[0] ;
mois = int_of_char str.[1] ;
heure = int_of_char str.[2] ;
minute = int_of_char str.[3] ;
seconde = int_of_char str.[4] } ;;
val decode : string -> horloge = <fun>

```

2. Écrire la **fonction** ou la classe de service en réutilisant un des serveurs génériques présentés. Ce service envoie les informations de la date à chaque connexion acceptée, puis referme la socket.

```

# main_serveur ;;
- : (in_channel -> out_channel -> 'a) -> unit = <fun>

# let horloge_service ic oc =
try
let date = Unix.localtime (Unix.time ()) in
let date_horloge =
{ jour = date.Unix.tm_mday ;
mois = date.Unix.tm_mon + 1 ;
heure = date.Unix.tm_hour ;
minute = date.Unix.tm_min ;
seconde = date.Unix.tm_sec } in

```

```

output_string oc (encode date_horloge) ;
flush oc
with exn -> print_endline "Fin du traitement"; flu

let main_horloge () = main_serveur horloge_service
val horloge_service : 'a -> out_channel -> unit = <fu
val main_horloge : unit -> unit = <fun>

```

3. Écrire un **client** qui effectue une mise à jour de sa date toutes les heures.

```

# main_client ;;
- : (in_channel -> out_channel -> 'a) -> unit = <fu

# let client_horloge ic oc =
  let date = ref { jour=0; mois=0; heure=0; minute=0
  try
    while true do
      let buffer = "xxxxx" in
      ignore (input ic buffer 0 5) ;
      date := decode buffer ;
      print_endline "BIP";
      flush stdout ;
      Unix.sleep 3600
    done
  with
    exn -> shutdown_connection ic ; raise exn ;;
val client_horloge : in_channel -> 'a -> unit = <fu

# let main_horloge () = main_client client_horloge
val main_horloge : unit -> unit = <fun>

```

4. Comment tenir **compte** du décalage de temps dû à la requête?

Une machine à café en réseau

On veut réaliser un petit service simulant un distributeur de boissons. La description sommaire du protocole d'échange entre un client et le service est la suivante :

- à sa connexion, le client reçoit la liste des boissons disponibles ;
- il envoie alors au serveur le nom de la boisson de son choix ;
- le serveur lui retourne le prix à payer ;
- le client envoie ce prix, ou toute autre somme ;
- le serveur répond par le nom de la boisson choisie et le montant de la monnaie rendue.

Le serveur peut aussi répondre par un message d'erreur s'il n'a pas compris une requête, n'a plus assez de monnaie, etc. Une requête d'un client ne contient toujours qu'un seul composant.

Les échanges se font sous forme de chaînes de caractères. Les différents composants des messages sont séparés par deux points et toute chaîne se termine par : `$\n`.

La fonction de service communique avec la machine à café en utilisant une file d'attente pour passer les commandes et une table de hachage pour récupérer les boissons préparées et la monnaie.

Cet exercice mettra en oeuvre la communication via des sockets, des processus légers avec un peu de concurrence et des objets.

1. Réécrire la fonction `establish_server` en utilisant les primitives de ThreadUnix.

```
# val hostaddr : string -> Unix.inet_addr = <fun>
val my_inet_addr : unit -> Unix.inet_addr = <fun>

let establish_server f saddr =
  let sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM
  Unix.bind sock saddr;
  Unix.listen sock 5;
  while true do
    let (s, _) = ThreadUnix.accept sock in
    let ic = Unix.in_channel_of_descr s
```

```

and oc = Unix.out_channel_of_descr s in
  ignore (Thread.create (f ic) oc)
done; ;
val establish_server :
  (in_channel -> out_channel -> 'a) -> Unix.sockaddr

```

2. Écrire deux fonctions, [get_request](#) et [send_answer](#). La première lit et déchiffre une requête et la seconde formate et envoie une réponse à partir d'une liste de chaînes de caractères.

```

# let read fd =
  let buf = String.create 1024 in
  let n = ThreadUnix.read fd buf 0 1024 in
  let s = String.sub buf 0 n in
  s ; ;
val read : Unix.file_descr -> string = <fun>

# let get_request fd =
  let s = read fd in
  match Str.split (Str.regexp "[:]") (String.sub s 0
    [s1] -> s1
    | _ -> failwith "BadRequestFormat" ; ;
val get_request : Unix.file_descr -> string = <fun>

# let write fd s =
  let leng = (String.length s) in
  let n = ThreadUnix.write fd s 0 leng in
  if n<leng then failwith "I/O error" ; ;
val write : Unix.file_descr -> string -> unit = <fun>

# let send_answer fd ss =
  let rec mk_answer = function
    [] -> ":\$\\n"
    | [s] -> s ^ ":\$\\n"
    | s::ss -> s ^ ":" ^ (mk_answer ss)
  in

```



```

    write fd (mk_answer ss) ;;
val send_answer : Unix.file_descr -> string list ->

# let send_cancel = let s = "cancel:$\n" in function
val send_cancel : Unix.file_descr -> unit = <fun>

```

3. Écrire une classe `cmd_fifo` pour gérer les commandes en attente. On attribuera un numéro unique à chaque nouvelle commande. Planter pour cela une classe `num_cmd_gen`.

```

# class cmd_fifo =
  object (self)
    val n = new num_cmd_gen
    val f = (Queue.create ()): (int*int*int) Queue.t
    val m = Mutex.create ()
    val c = Condition.create ()

    method add num_drink paid =
      let num_cmd = n#get() in
      Mutex.lock m ;
      Queue.add (num_cmd, num_drink, paid) f ;
      Mutex.unlock m ;
      Condition.signal c ;
      num_cmd

    method wait () =
      Mutex.lock m ;
      Condition.wait c m ;
      let cmd = Queue.take f in
      Mutex.unlock m ;
      cmd
    end ;;
class cmd_fifo :
  object
    val c : Condition.t
    val f : (int * int * int) Queue.t
    val m : Mutex.t
    val n : num_cmd_gen

```

```

method add : int -> int -> int
method wait : unit -> int * int * int
end

# class num_cmd_gen =
  object
    val mutable x = 0
    val m = Mutex.create ()

    method get() =
      Mutex.lock m ;
      x <- x+1 ;
      let r = x in
      Mutex.unlock m ;
      r
    end ;;
  class num_cmd_gen :
    object val m : Mutex.t val mutable x : int method

```

4. Écrire une classe [ready_table](#) pour stocker les boissons préparées par la machine.

```

# class ready_table size =
  object
    val t = (Hashtbl.create size : (int, (string * int)
    val m = Mutex.create ()
    val c = Condition.create ()

    method add num_cmd num_drink change =
      Mutex.lock m ;
      Hashtbl.add t num_cmd (num_drink, change) ;
      Mutex.unlock m ;
      Condition.broadcast c

    method wait num_cmd =
      Mutex.lock m;
      while not (Hashtbl.mem t num_cmd) do Condition.wait

```

```

let cmd = Hashtbl.find t num_cmd in
Hashtbl.remove t num_cmd ;
Mutex.unlock m ;
cmd
end ;;
class ready_table :
  int ->
  object
  val c : Condition.t
  val m : Mutex.t
  val t : (int, string * int) Hashtbl.t
  method add : int -> string -> int -> unit
  method wait : int -> string * int
end

```

- Écrire la classe **machine** qui modélise la machine à café. Elle possédera une méthode **run** bouclant sur la séquence : attendre une commande puis la préparer, tant qu'il reste des boissons disponibles. On définira un type **drink_descr** indiquant, pour chaque boisson : son nom, la quantité en stock, la quantité restant après satisfaction des commandes et le prix. On utilisera également une fonction auxiliaire **array_index** qui donne l'indice du premier élément d'un tableau satisfaisant un critère passé en paramètre.

```

# class machine (f_cmd0:cmd_fifo) (t_ready0:ready_t
object (self)
  val f_cmd = f_cmd0
  val t_ready = t_ready0
  val mutable nb_available_drinks = 0
  val drinks_table =
    [| { name="cafe"; real_stock=10; virtual_stock=10;
      { name="the"; real_stock=5; virtual_stock=5; price
      { name="chocolat"; real_stock=10; virtual_stock=10
  val mutable cash = 0
  val m = Mutex.create()

initializer nb_available_drinks <- Array.length dr

```

```

method get_drink_price i = drinks_table.(i).price
method get_drink_index s = array_index drinks_table

method get_menu () =
let f d ns = if d.real_stock > 0 then d.name::ns else []
Array.fold_right f drinks_table []

method cancel_cmd num_drink =
let drink = drinks_table.(num_drink) in
drink.virtual_stock <- drink.virtual_stock+1

method set_cmd num_drink paid = f_cmd#add num_drink paid

method wait_cmd num_cmd = t_ready#wait num_cmd

method deliver_drink num_drink =
let drink = drinks_table.(num_drink) in
drink.real_stock <- drink.real_stock-1 ;
if drink.real_stock = 0 then nb_available_drinks <- nb_available_drinks-1

method run() =
while nb_available_drinks>0 do
let (num_cmd, num_drink, amount) = f_cmd#wait () in
let drink = drinks_table.(num_drink) in
let change = amount - drink.price in
Mutex.lock m ;
if (drink.virtual_stock > 0) & (cash >= change)
then
begin
drink.virtual_stock <- drink.virtual_stock-1 ;
cash <- cash + drink.price ;
t_ready#add num_cmd drink.name change
end
else t_ready#add num_cmd "cancel" 0 ;
Mutex.unlock m
done
end ;;

class machine :
cmd_fifo ->

```

```

ready_table ->
object
val mutable cash : int
val drinks_table : drink_descr array
val f_cmd : cmd_fifo
val m : Mutex.t
val mutable nb_available_drinks : int
val t_ready : ready_table
method cancel_cmd : int -> unit
method deliver_drink : int -> unit
method get_drink_index : string -> int
method get_drink_price : int -> int
method get_menu : unit -> string list
method run : unit -> unit
method set_cmd : int -> int -> int
method wait_cmd : int -> string * int
end

```

```

# type drink_descr =
{ name : string;
mutable real_stock : int;
mutable virtual_stock : int;
price : int } ;;

```

```

# let array_index t f =
let i = ref 0 in
let n = Array.length t in
while (!i < n) & (not (f t.(!i))) do incr i done ;
if !i=n then raise Not_found else !i ;;
val array_index : 'a array -> ('a -> bool) -> int =

```

6. Écrire une fonction de service [waiter](#).

```

# let waiter mach ic oc =
let f_in = Unix.descr_of_in_channel ic in
let f_out = Unix.descr_of_out_channel oc in
(try

```

```

send_answer f_out (mach#get_menu()) ;
let drink_name = get_request f_in in
let num_drink = mach#get_drink_index drink_name in
let drink_price = mach#get_drink_price num_drink in
send_answer f_out [string_of_int drink_price] ;
let paid = int_of_string (get_request f_in) in
if paid < drink_price then failwith "NotEnough" ;
let num_cmd = mach#set_cmd num_drink paid in
let drink_name, change = mach#wait_cmd num_cmd in
mach#deliver_drink num_drink;
send_answer f_out [drink_name; (string_of_int chan
with
Not_found -> send_cancel f_out
| Failure("int_of_string") -> send_cancel f_out
| Failure("I/O error") -> send_cancel f_out
| Failure("NotEnough") -> send_cancel f_out
| Failure("BadRequestFormat") -> send_cancel f_out
);
close_in ic ;
flush oc ;
close_out oc ;
Thread.exit () ;;
val waiter :
< deliver_drink : 'a -> 'b; get_drink_index : stri
get_drink_price : 'a -> int; get_menu : unit -> st
set_cmd : 'a -> int -> 'c; wait_cmd : 'c -> string
in_channel -> out_channel -> unit = <fun>

```

7. Écrire une fonction principale **main** qui récupère le numéro de port du service sur la ligne de commande et procède aux diverses initialisations. En particulier, la machine à café est exécutée par un processus.

```

# let main () =
if Array.length Sys.argv < 2
then
begin
Printf.eprintf "usage : %s port\n" Sys.argv.(0) ;
exit 1

```

```

end
else
begin
let port = int_of_string Sys.argv.(1) in
let f_cmd = new cmd_fifo in
let t_ready = new ready_table in
let mach = new machine f_cmd (t_ready 13) in
ignore (Thread.create mach#run ()) ;
establish_server (waiter mach) (Unix.ADDR_INET (my
end ;;
val main : unit -> unit = <fun>

```

Résumé

Ce chapitre a présenté les nouvelles possibilités offertes par la programmation répartie. La communication entre programmes s'effectue à partir du mécanisme de base que sont les sockets (prises de communication) issues des protocoles de bas niveau du réseau Internet. La manière la plus classique de construire une application distribuée est le modèle client-serveur. Les schémas d'actions entre le client et le serveur sont asymétriques. La communication repose sur un protocole le plus souvent en texte clair. La programmation fonctionnelle et la programmation par objets permettent de réaliser facilement de telles applications réparties. Le modèle client-serveur se prête à différentes architectures logicielles, à 2 ou 3 niveaux, selon la répartition des tâches entre eux.

Pour en savoir plus

La communication entre programmes Objective CAML distants peut être riche. L'utilisation de protocoles texte est grandement facilitée par les outils d'analyse syntaxique du langage (voir chapitre 11). Le mécanisme de persistance apporté par le module Marshal (voir chapitre 8) permet de transporter en format interne des données complexes y compris des valeurs fonctionnelles si les deux programmes communicants sont les mêmes. Il manque principalement la transmission d'instances de classe. Une solution à ce problème est d'utiliser un ORB (Object Request Broker) pour

transmettre des objets ou invoquer des méthodes distantes. Cette architecture existe déjà pour de nombreux langages objets sous la forme de la norme CORBA (Common ORB Architecture). Cette norme, du groupe OMG, dont la première mouture date de 1990, permet d'utiliser des objets distants en étant indépendants du langage d'implantation des classes.

Lien: <http://www.omg.org>

Les deux fonctions principales sont de pouvoir envoyer des objets à distance et surtout de pouvoir référencer de plusieurs endroits du réseau un même objet, pour invoquer ses méthodes permettant de modifier ses variables d'instances. De plus cette norme est indépendante des langages d'implantation de ces objets distants. Pour cela, un ORB fournit un langage de description des interfaces appelé IDL (Interface Declaration Language), à la manière de CAMLIDL pour l'interface entre Objective CAML et C. Pour l'instant, il n'y a pas d'ORB intégrant le langage Objective CAML, mais il n'y a pas d'impossibilité de réalisation car le langage IDL est un sous-ensemble des langages objets à classes. Pour simplifier CORBA fournit un bus logiciel (IIOP) qui permet le transfert et l'adressage de données distantes.

Le fait de pouvoir référencer le même objet de plusieurs points du réseau simule une mémoire partagée distribuée, ce qui n'est pas sans poser de problèmes pour les récupérateurs automatiques de mémoire.

Le fait de référencer un objet distant ne provoque pas le transfert de code. On ne peut recevoir une copie d'une instance d'une classe que si la classe existe dans le serveur. Or pour certaines applications client-serveur, il peut être nécessaire d'effectuer un chargement dynamique de code (applets en Java) et même de faire migrer des processus avec leur code. Un exemple intéressant de chargement dynamique de code distant est le navigateur MMM réalisé en Objective CAML par François Rouaix :

Lien: <http://caml.inria.fr/~rouaix/mmm/>

Ce navigateur permet de consulter le Web, mais aussi de télécharger des applets Objective CAML d'un serveur pour les exécuter dans sa

fenêtre graphique.

Applications

Introduction

La première application est en fait une boîte à outils qui permet de réaliser rapidement des applications client-serveur pour communiquer des valeurs Objective CAML. Il suffit en fait d'implanter les fonctions de sérialisation de ces valeurs, puis d'appliquer un foncteur pour obtenir une classe abstraite du serveur dont on pourra implanter la fonction de traitement via un héritage.

La deuxième application reprend la simulation des robots, présentée à la page ??, pour l'étendre au modèle client-serveur. Le monde devient le serveur où évoluent les robots clients. On simule ainsi une mémoire distribuée partagée par l'ensemble des clients et se trouvant potentiellement sur différentes machines du réseau.

La troisième application est une implantation de petits serveurs HTTP (appelés servlets). Un serveur sait répondre à une requête HTTP comme l'accès à une page HTML. De plus il est possible de passer des valeurs à ces requêtes, respectant le format CGI des serveurs HTTP. On utilise immédiatement cette fonctionnalité pour construire un serveur de requêtes pour la gestion d'associations, décrite à la page ?. On utilise comme client un navigateur WEB à qui est transmis une première page contenant le formulaire d'interrogation.

Boîte à outils client-serveur

Nous présentons un ensemble de modules pour la construction de client-serveur entre programmes Objective CAML. Cette boîte à outils est ensuite utilisée dans les deux applications suivantes.

Une application se distingue d'une autre par le protocole qu'elle utilise et par les traitements qu'elle y associe. Pour le reste, à savoir les mécanismes d'attente de connexion, de détachement du traitement de la connexion sur un autre processus, les lectures et

écritures sur une socket, les applications sont très semblables les unes aux autres.

Profitant de la possibilité de mélanger la généricité modulaire et l'extension des objets que nous offre Objective CAML, nous allons réaliser un ensemble de foncteurs prenant comme argument un protocole de communication et engendrant des classes génériques implantant les mécanismes des clients et des serveurs. Il ne nous restera ensuite qu'à les sous-classer pour obtenir des traitements particuliers.

Protocoles

Un protocole de communication est un type de données qu'il est possible de traduire sous forme de chaînes de caractères afin de faire transiter d'une machine à une autre des données par une socket. Ceci peut se traduire sous la forme d'une signature.

```
# module type PROTOCOL =  
  sig  
    type t  
    val to_string : t -> string  
    val of_string : string -> t  
  end ;;
```

La signature impose que le type de données soit monomorphe, mais hormis cette restriction, du moment qu'il est possible de le traduire en chaîne de caractères et inversement, nous pouvons choisir comme structure de données des valeurs aussi complexes que l'on souhaite. En particulier, rien ne nous interdit d'avoir comme donnée un objet.

```
# module Integer =  
  struct  
    class integer x =  
      object  
        val v = x  
        method x = v  
        method str = string_of_int v  
      end
```

```

type t = integer
let to_string o = o#str
let of_string s = new integer (int_of_string s)
end ;;

```

En faisant quelques restrictions sur les types de données manipulables, nous pouvons utiliser le module Marshal, décrit page ??, pour définir les fonctions de traduction.

```

# module Make_Protocole = functor ( T : sig type t end )
struct
  type t = T.t
  let to_string (x:t) = Marshal.to_string x [Marshal.Clos
  let of_string s = (Marshal.from_string s 0 : t)
end ;;

```

Communication

Puisqu'un protocole est une donnée qu'il est possible de traduire sous la forme d'une chaîne de caractères, nous pouvons en faire un persistant et le stocker dans un fichier.

La seule difficulté pour lire une valeur depuis un fichier quand on ne connaît pas son type est qu'a priori nous ne connaissons pas la taille de la donnée en question. Et puisque le fichier en question sera en fait une socket, nous ne pouvons pas nous fier au marqueur de fin de fichier. Pour résoudre ce problème, nous faisons précéder la donnée par la taille en nombre de caractères à lire. Les douze premiers caractères contiennent sa taille et des espaces.

Le foncteur Com prend en paramètre un module de signature PROTOCOL et définit les fonctions d'émission et de réception des valeurs codées dans le protocole.

```

# module Com = functor (P : PROTOCOL) ->
struct
  let send fd m =
  let mes = P.to_string m in

```

```

let l = (string_of_int (String.length mes)) in
let buffer = String.make 12 ' ' in
for i=0 to (String.length l)-1 do buffer.[i] <- l.[i] done ;
ignore (ThreadUnix.write fd buffer 0 12) ;
ignore (ThreadUnix.write fd mes 0 (String.length mes))

let receive fd =
let buffer = String.make 12 ' '
in
ignore (ThreadUnix.read fd buffer 0 12) ;
let l = let i = ref 0
in while (buffer.[!i]<>' ') do incr i done ;
int_of_string (String.sub buffer 0 !i)
in
let buffer = String.create l
in ignore (ThreadUnix.read fd buffer 0 l) ;
P.of_string buffer
end ;;

module Com :
  functor (P : PROTOCOL) ->
    sig
      val send : Unix.file_descr -> P.t -> unit
      val receive : Unix.file_descr -> P.t
    end
end

```

Notons que nous utilisons les fonctions read et write du module ThreadUnix et non celles du module Unix; cela nous permettra d'utiliser les fonctions du module dans un thread sans bloquer l'exécution des autres processus.

Serveur

Un serveur est réalisé comme une classe abstraite paramétrée par le type de données du protocole. Son constructeur prend comme argument le numéro du port et le nombre de connexions simultanées acceptables. La méthode de traitement d'une requête est abstraite; elle doit être implantée dans une sous-classe de server pour obtenir une classe concrète.

```

# module Server = functor (P : PROTOCOL) ->
struct
module Com = Com (P)

class virtual ['a] server p np =
object (s)
constraint 'a = P.t
val port_num = p
val nb_pending = np
val sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM

method start =
let host = Unix.gethostname (Unix.gethostname()) in
let h_addr = host.Unix.h_addr_list.(0) in
let sock_addr = Unix.ADDR_INET(h_addr, port_num) in
Unix.bind sock sock_addr ;
Unix.listen sock nb_pending ;
while true do
let (service_sock, client_sock_addr) = ThreadUnix.accept sock in
ignore (Thread.create s#treat service_sock)
done
method send = Com.send
method receive = Com.receive
method virtual treat : Unix.file_descr -> unit
end
end ;;

```

Afin de fixer les idées, nous reprenons le service majuscule comme illustration mais en donnant la possibilité d'envoyer des listes de mots.

```

# type message = Str of string | LStr of string list ;;
# module Maj_Protocol = Make_Protocol (struct type t=me
# module Maj_Server = Server (Maj_Protocol) ;;

# class maj_server p np =
object (self)
inherit [message] Maj_Server.server p np
method treat fd =

```

```

match self#receive fd with
  Str s  -> self#send fd (Str (String.uppercase s)) ;
  Unix.close fd
| LStr l -> self#send fd (LStr (List.map String.uppercase l)) ;
  Unix.close fd
end ;;

class maj_server :
  int ->
  int ->
  object
    val nb_pending : int
    val port_num : int
    val sock : Unix.file_descr
    method receive : Unix.file_descr -> Maj_Protocol.t
    method send : Unix.file_descr -> Maj_Protocol.t -> unit
    method start : unit
    method treat : Unix.file_descr -> unit
  end
end

```

Le traitement se décompose en la réception de la requête, son filtrage, son traitement et l'émission du résultat. Le foncteur permet de se concentrer sur le service pour réaliser le serveur, le reste est générique. Cependant, si on souhaite avoir un mécanisme différent, comme par exemple gérer des acquittements, rien n'interdit de redéfinir les méthodes de communication héritées.

Client

Pour réaliser des clients utilisant un protocole donné, nous définissons trois fonctions généralistes :

- `connect` : établit une connexion avec un serveur, elle prend son adresse (adresse IP et numéro de port) et rend un descripteur de fichier correspondant à une socket connectée au serveur.
- `emit_simple` : ouvre une connexion, envoie un message et referme la connexion.
- `emit_answer` : idem que la précédente, mais attend la réponse du serveur avant de refermer la connexion.

```

# module Client = functor (P : PROTOCOL) ->
  struct
    module Com = Com (P)

    let connect addr port =
    let sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM
    and in_addr = (Unix.gethostbyname addr).Unix.h_addr_list[0]
    in ThreadUnix.connect sock (Unix.ADDR_INET(in_addr, port))
    sock

    let emit_simple addr port mes =
    let sock = connect addr port
    in Com.send sock mes ; Unix.close sock

    let emit_answer addr port mes =
    let sock = connect addr port
    in Com.send sock mes ;
    let res = Com.receive sock
    in Unix.close sock ; res
    end ;;
module Client :
  functor(P : PROTOCOL) ->
    sig
      module Com :
        sig
          val send : Unix.file_descr -> P.t -> unit
          val receive : Unix.file_descr -> P.t
        end
        val connect : string -> int -> Unix.file_descr
        val emit_simple : string -> int -> P.t -> unit
        val emit_answer : string -> int -> P.t -> P.t
      end

```

Les deux dernières fonctions sont de plus haut niveau que la première. Le vecteur de la liaison entre le client et le serveur n'apparaît pas. L'utilisateur de `emit_answer` n'a même pas besoin de savoir que le calcul qu'il demande est effectué sur une machine distante. Pour lui, il invoque une fonction qui est représentée par une adresse et un port avec un argument qui est le message envoyé,

et une valeur lui est retournée. Le côté distribué peut lui paraître anecdotique.

Un client du service majuscule est excessivement aisé à réaliser. En supposant que la machine boulmich héberge ce service sur le port 12345; la fonction `list_uppercase` peut se définir par un appel au service.

```
# let list_uppercase l =  
  let module Maj_client = Client (Maj_Protocol)  
  in match Maj_client.emit_answer "boulmich" 12345 (LStr  
    with Str x -> [x]  
      | LStr x -> x ;;  
val list_uppercase : string list -> string list = <fun>
```

Pour en faire plus

La première amélioration à apporter à notre boîte à outils est une gestion des erreurs qui ici est totalement absente. Une récupération des exceptions qui surviennent lors de la rupture d'une connexion et un mécanisme de `< >` seraient les bienvenus.

Dans la même veine, le client et le serveur gagneraient à être munis d'un mécanisme de timeout permettant de borner le temps d'attente d'une réponse.

Le fait d'avoir réalisé le serveur générique comme une classe, qui de surcroît est paramétrée par le type de données qui transitent sur le réseau, permet de l'étendre facilement pour augmenter ou modifier son comportement afin de réaliser les améliorations souhaitées.

Une autre approche est d'enrichir les protocoles de communication. On peut par exemple ajouter des requêtes d'acquiescement au protocole, ou encore accompagner chaque requête d'un checksum permettant de vérifier que le réseau n'a pas corrompu les données.

Les robots de l'aube

Comme nous l'avions promis dans la dernière application de la troisième partie (page ??), nous reprenons le problème des robots pour le traiter dans un cadre distribué où le monde est un serveur et où chaque robot est un processus indépendant pouvant s'exécuter depuis une machine distante.

Cette application est un bon résumé des possibilités du langage Objective CAML puisque nous allons y exploiter et mélanger la plupart de ses traits. Outre le modèle distribué qui nous est imposé par l'exercice, nous avons recours à la concurrence pour réaliser le serveur de sorte que les multiples connexions soient traitées indépendamment tout en s'appuyant sur une unique représentation mémoire du « monde ». L'accès et la modification de l'état des cases du monde devront en conséquence être protégés par des sections critiques.

Afin de réutiliser au maximum le code qui a déjà été réalisé pour les robots d'une part, et les architectures client-serveur d'autre part, nous employons de façon conjointe foncteurs et héritage de classes.

Cette application est assez minimaliste, mais nous verrons que son architecture se prête particulièrement bien à des extensions dans de multiples directions.

Monde-Serveur

Nous prenons une représentation du monde similaire à celle que nous avons développée dans la partie III. Le monde est une grille de taille finie et chaque case de cette grille ne peut être occupée que par un seul robot. Un robot est connu par son nom et par sa position, le monde est déterminé par sa taille et par les robots qui y séjournent. Ces différentes informations sont représentées par les types suivants :

```
# type position = { x:int ; y:int } ;;

# type robot_info = { name : string ; mutable pos : position } ;;
type world_info = { length : int ; width : int ;
mutable robots : robot_info list } ;;
```

Le monde aura à servir deux sortes de clients :

- des clients passifs qui se contentent d'espier les déplacements des différents robots. Ils nous serviront pour réaliser les clients chargés des affichages. Nous les appellerons des espions.
- des clients actifs, susceptibles de demander au serveur de les déplacer et donc de modifier son état.

Ces deux catégories de clients et leur comportement vont déterminer l'ensemble des messages qu'échangeront serveur et clients.

Un client en se connectant se déclare passif (Spy) ou actif (Enter). Un espion reçoit en réponse à sa connexion l'état global du monde. Ensuite, il est tenu informé de tous les changements. Par contre, il ne peut pas soumettre de requête. Un robot qui se connecte doit donner ses caractéristiques (son nom et sa position initiale souhaitée), le monde lui confirme alors son arrivée. Ensuite, il pourra demander des informations : sa propre position (GetPos) ou la liste des robots qui l'entourent (Look). Il peut aussi solliciter le monde pour se déplacer. Le protocole de requêtes du monde des robots distribués est représenté par le type suivant :

```
# type query =  
| Spy                                (* requêtes de déclaration ini  
| Enter of robot_info  
  
| Move of position                  (* requêtes des robots *)  
| GetPos  
| Look of int  
  
| World of world_info              (* messages délivrés par le mo  
| Pos of robot_info  
| Exit of robot_info ;;
```

De ce protocole, et avec les foncteurs de la << boîte à outils distribuée >> du chapitre précédent, on tire immédiatement le serveur générique.

```
# module Pquery = Make_Protocol (struct type t = query
```

```
# module Squery = Server (Pquery) ;;
```

Il ne nous reste plus qu'à préciser le comportement du serveur en implantant la méthode `treat` et en y incorporant les données qui, d'une part, représentent le monde et, d'autre part, permettent la gestion des connexions.

Plus précisément, le serveur possède une variable `world` (de type *world_info*) qui est protégée par le verrou `sem` (de type *Mutex.t*). Il possède aussi une variable `spies` qui est une liste de files de messages à envoyer aux observateurs, à raison d'une file par espion. Pour réveiller les processus chargés de l'envoi de ces messages, le serveur dispose aussi d'un signal (de type *Condition.t*).

Nous donnons une fonction auxiliaire `dist` de calcul de distance entre deux positions :

```
# let dist p q = max (abs (p.x-q.x)) (abs (p.y-q.y)) ;;  
val dist : position -> position -> int = <fun>
```

La fonction `critical` encapsule dans une section critique un calcul de valeur :

```
# let critical m f a =  
  Mutex.lock m ; let r = f a in Mutex.unlock m ; r ;;  
val critical : Mutex.t -> ('a -> 'b) -> 'a -> 'b = <fun>
```

Voici la définition de la classe `server` implantant le monde-serveur. Elle est longue, mais nous en donnons ci-après une explication synthétique.

```
# class server l w n np =  
  object (self)  
    inherit [query] Squery.server n np  
    val world = { length=l ; width=w ; robots=[] }  
    val sem = Mutex.create ()  
    val mutable spies = []  
    val signal = Condition.create ()  
  
    method lock = Mutex.lock sem  
    method unlock = Mutex.unlock sem
```

```

method legal_pos p = p.x>=0 && p.x<1 && p.y>=0 && p.y<w

method free_pos p =
let is_not_here r = r.pos.x<>p.x || r.pos.y<>p.y
in critical sem (List.for_all is_not_here) world.robots

method legal_move r p =
let dist1 p = (dist r.pos p) <= 1
in (critical sem dist1 p) && self#legal_pos p && self#f

method queueing_message mes =
List.iter (Queue.add mes) spies ;
Condition.broadcast signal

method trace_loop s q =
let foo = Mutex.create () in
let f () =
try
spies <- q :: spies ;
self#send s (World world) ;
while true do
while Queue.length q = 0 do Condition.wait signal foo do
self#send s (Queue.take q)
done
with _ -> spies <- List.filter ((!=) q) spies ;
Unix.close s
in ignore (Thread.create f ())

method remove_robot r =
self#lock ;
world.robots <- List.filter ((<>) r) world.robots ;
self#queueing_message (Exit {r with name=r.name}) ;
self#unlock

method try_move_robot r p =
if self#legal_move r p
then begin
self#lock ;

```

```

r.pos <- p ;
self#queueing_message (Pos {r with name=r.name}) ;
self#unlock
end

method treat_robot s r =
let f () =
try
world.robots <- r :: world.robots ;
self#send s (Pos r) ;
self#queueing_message (Pos r) ;
while true do
Thread.delay 0.5 ;
match self#receive s with
Move p -> self#try_move_robot r p
| GetPos -> self#send s (Pos r)
| Look d ->
self#lock ;
let dist p = max (abs (p.x-r.pos.x)) (abs (p.y-r.pos.y))
let l = List.filter (fun x -> (dist x.pos)<=d) world.robots
in self#send s (World { world with robots = l }) ;
self#unlock
| _ -> ()
done
with _ -> self#unlock ;
self#remove_robot r ;
Unix.close s
in ignore (Thread.create f ())

method treat s =
match self#receive s with
Spy -> self#trace_loop s (Queue.create ())
| Enter r ->
( if not (self#legal_pos r.pos && self#free_pos r.pos)
let i = ref 0 and j = ref 0 in
( try
for x=0 to l do
for y=0 to w do
let p = { x=x ; y=y }
in if self#legal_pos p && self#free_pos p

```

```

then ( i:=x ; j:=y; failwith "treat" )
done done ;
Unix.close s
with Failure "treat" -> r.pos <- { x= !i ; y= !j } ))
self#treat_robot s r
| _ -> Unix.close s

end ;;
class server :
  int ->
  int ->
  int ->
  int ->
  object
  val nb_pending : int
  val port_num : int
  val sem : Mutex.t
  val signal : Condition.t
  val sock : Unix.file_descr
  val mutable spies : Pquery.t Queue.t list
  val world : world_info
  method free_pos : position -> bool
  method legal_move : robot_info -> position -> bool
  method legal_pos : position -> bool
  method lock : unit
  method queueing_message : Pquery.t -> unit
  method receive : Unix.file_descr -> Pquery.t
  method remove_robot : robot_info -> unit
  method send : Unix.file_descr -> Pquery.t -> unit
  method start : unit
  method trace_loop : Unix.file_descr -> Pquery.t Queue.t
  method treat : Unix.file_descr -> unit
  method treat_robot : Unix.file_descr -> robot_info -> unit
  method try_move_robot : robot_info -> position -> unit
  method unlock : unit
end

```

La méthode `treat` s'emploie dès le départ à distinguer la nature du client. Suivant qu'il est actif ou passif, c'est un traitement particulier qui est appelé : `trace_loop` pour un observateur, `treat_robot` pour un

robot. Dans le second cas, on vérifie que la position initiale proposée par le client est compatible avec l'état du monde, sinon on tâche de lui trouver une position initiale valide. Le reste du code peut se partager en quatre catégories :

1. **Les méthodes générales** : ce sont celles que nous avons pour les mondes généraux de la partie III. Principalement, il s'agit de vérifier qu'un déplacement est légal pour un robot donné.
2. **La gestion des observateurs** : chaque observateur est associé à une socket par laquelle lui sont envoyées les données, à une queue contenant tous les messages qui ne lui ont pas encore été envoyés et à un processus. La méthode `trace_loop` est une boucle sans fin qui vide la queue des messages en les émettant et qui s'endort quand elle est vide. Le remplissage est effectué sur toutes les files d'attente par la méthode `queueing_message`. Notons qu'après avoir ajouté un message, le signal de réveil est envoyé à tous les processus.
3. **La gestion des robots** : là encore, chaque robot est associé à un processus qui lui est dédié. La méthode `treat_robot` est une boucle sans fin : elle attend la réception d'une requête, la traite et y répond s'il y a lieu. Puis elle se remet en attente de la prochaine requête. Notons que ce sont les méthodes gérant le robot qui font les appels à la méthode `queueing_message` quand une modification de l'état du monde a eu lieu. Si la connexion avec le robot est perdue, c'est à dire si une exception est levée pendant la réception des requêtes, le robot est considéré comme se retirant et son départ est signalé aux observateurs.
4. **Les méthodes héritées** : ce sont celles du serveur générique obtenu par application du foncteur `Server` au protocole de notre application.

Client-Observateur

Le foncteur `Client` nous donne les fonctions génériques de connexion avec un serveur selon le protocole particulier qui nous

occupe ici.

```
# module Cquery = Client (Pquery) ;;
module Cquery :
  sig
    module Com :
      sig
        val send : Unix.file_descr -> Pquery.t -> unit
        val receive : Unix.file_descr -> Pquery.t
      end
      val connect : string -> int -> Unix.file_descr
      val emit_simple : string -> int -> Pquery.t -> unit
      val emit_answer : string -> int -> Pquery.t -> Pquery.t
    end
```

Le comportement de l'espion est simple : il se connecte au serveur et affiche les informations que celui-ci lui transmet. L'espion dispose de trois fonctions d'affichage que nous donnons ci-dessous :

```
# let display_robot r =
  Printf.printf "Le robot %s se trouve en (%d,%d)\n" r.name
  flush stdout ;;
val display_robot : robot_info -> unit = <fun>

# let display_exit r =   Printf.printf "Le robot %s se r
  flush stdout ;;
val display_exit : robot_info -> unit = <fun>

# let display_world w =
  Printf.printf "Le monde est un damier de %d par %d \n"
  List.iter display_robot w.robots ;
  flush stdout ;;
val display_world : world_info -> unit = <fun>
```

La fonction principale du client-espion est :

```
# let trace_client name port =
  let sock = Cquery.connect name port
  in Cquery.Com.send sock Spy ;
  ( match Cquery.Com.receive sock with
    World w -> display_world w
```



```

| _ -> failwith "le serveur ne suit pas le protocole" )
while true do
  match Cquery.Com.receive sock with
  Pos r -> display_robot r
  | Exit r -> display_exit r
  | _ -> failwith "le serveur ne suit pas le protocole"
done ;;
val trace_client : string -> int -> unit = <fun>

```

Il y a deux façons de procéder pour réaliser un affichage graphique. La première est simple mais peu économique : puisqu'en début de connexion le serveur envoie la totalité de l'information, on peut se contenter d'ouvrir une nouvelle connexion à intervalles réguliers, d'afficher le monde dans sa globalité et de refermer la connexion. L'autre possibilité consiste à utiliser les informations envoyées par le serveur pour maintenir une copie de l'état du monde. Il est alors aisé de n'afficher que les modifications d'état au fil de la réception des messages. C'est cette seconde solution que nous avons mise en oeuvre.

Client-Robot

Tels que nous les avons définis dans le précédent chapitre (cf. page [??](#)), les robots suivaient la signature suivante.

```

# module type ROBOT =
  sig
    class robot : int -> int ->
    object
      val mutable i : int
      val mutable j : int
      method get_pos : int * int
      method next_pos : unit -> int * int
      method set_pos : int * int -> unit
    end
  end ;;

```

La partie que nous souhaitons conserver des différentes classes est celle qui justement variait d'un type de robot à l'autre et qui définissait son comportement : la méthode `next_pos`.

De plus nous avons besoin d'une méthode pour connecter le robot à un monde (start) et d'une boucle alternant calcul d'une nouvelle position et communication avec le serveur pour soumettre la position choisie.

Nous définissons un foncteur qui partant d'une classe implantant un robot virtuel, c'est-à-dire respectant la signature ROBOT, crée, par héritage, une nouvelle classe contenant les méthodes propres à faire du robot un client autonome.

```
# module RobotClient (R : ROBOT) =  
  struct  
    class robot robname x y hostname port =  
      object (self)  
        inherit R.robot x y as super  
        val mutable socket = Unix.stderr  
        val mutable rob = { name=robname ; pos={x=x;y=y} }  
  
        method private adjust_pos r =  
          rob.pos <- r.pos ; i <- r.pos.x ; j <- r.pos.y  
  
        method get_pos =  
          Cquery.Com.send socket GetPos ;  
          match Cquery.Com.receive socket with  
          Pos r -> self#adjust_pos r ; super#get_pos  
          | _ -> failwith "le serveur ne respecte pas le protocol"  
  
        method set_pos =  
          failwith "la méthode set_pos ne doit pas être utilisée"  
  
        method start =  
          socket <- Cquery.connect hostname port ;  
          Cquery.Com.send socket (Enter rob) ;  
          match Cquery.Com.receive socket with  
          Pos r -> self#adjust_pos r ; self#run  
          | _ -> failwith "le serveur ne respecte pas le protocol"  
  
        method run =  
          while true do  
            let (x,y) = self#next_pos ()
```

```

in Cquery.Com.send socket (Move {x=x;y=y}) ;
ignore (self#get_pos)
done
end
end ;;
module RobotClient :
  functor(R : ROBOT) ->
    sig
      class robot :
        string ->
        int ->
        int ->
        string ->
        int ->
        object
          val mutable i : int
          val mutable j : int
          val mutable rob : robot_info
          val mutable socket : Unix.file_descr
          method private adjust_pos : robot_info -> unit
          method get_pos : int * int
          method next_pos : unit -> int * int
          method run : unit
          method set_pos : int * int -> unit
          method start : unit
        end
      end
    end
  end
end

```

Remarquons que la méthode `get_pos` a été redéfinie comme une interrogation auprès du serveur car les variables d'instance `i` et `j` ne sont pas fiables puisqu'elles peuvent être modifiées sans l'accord du monde. Pour les mêmes raisons, l'emploi de `set_pos` a été invalidé de sorte que son appel déclenche systématiquement une exception. Ce traitement peut apparaître brutal, mais il y a fort à parier que si cette méthode est utilisée par `next_pos` alors un décalage entre la position réelle (celle connue par le serveur) et la position supposée (celle connue par le client) apparaîtra.

Nous utilisons le foncteur `RobotClient` pour créer les différentes classes correspondant aux différents robots.

```
# module Fix = RobotClient (struct class robot = fix_robot
# module Crazy = RobotClient (struct class robot = crazy
# module Obstinate = RobotClient (struct class robot = o
```

Le petit programme suivant permet de lancer depuis une ligne de commande, le serveur ou les différents clients. Chacun d'eux est identifié par l'argument passé au programme.

```
# let port = 1200 in
if Array.length Sys.argv >=2 then
match Sys.argv.(1) with
| "1" -> let s = new server 25 30 port 10 in s#start
| "2" -> trace_client "localhost" port
| "3" -> let o = new Fix.robot "fixe" 10 10 "localhost"
| "4" -> let o = new Crazy.robot "fou" 10 10 "localhost"
| "5" -> let o = new Obstinate.robot "obstine" 10 10 "1
in o#start
| _ -> () ;;
```

Pour en faire plus

Le monde des robots a l'imagination fertile. Avec les éléments déjà donnés ici, on peut facilement réaliser un « robot intelligent » qui soit à la fois robot et espion. Cela permettrait de faire coopérer les différents habitants du monde. On peut alors étendre l'application pour obtenir un petit jeu d'action comme « poules-renards-vipères » dans lequel les renards chassent les poules, les vipères chassent les renards et les poules mangent les vipères.

Servlets HTTP

Un servlet est un « module » à intégrer dans une application serveur pour répondre aux requêtes des clients. Bien qu'un servlet ne soit pas spécifique à un protocole, on utilisera le protocole HTTP pour la communication (voir la figure 21.1). Dans la pratique le terme servlet correspond à un servlet HTTP.

Le moyen classique de construire des pages HTML dynamiques sur un serveur HTTP est d'utiliser des commandes CGI (Common Gateway Interface). Celles-ci prennent en argument une URL pouvant contenir des données provenant d'un formulaire. L'exécution produit alors une page HTML qui est envoyée au client. On trouvera aux liens suivants la description du protocole HTTP et des CGI.

Lien: <http://www.eisti.fr/eistiweb/docs/normes/rfc1945/1945tm.htm>

Lien: <http://hoohoo.ncsa.uiuc.edu/docs/cgi/overview.html>

C'est un mécanisme un peu lourd car il lance à chaque requête un nouveau programme.

Les servlets HTTP sont lancés une fois pour toutes, et peuvent décoder les arguments du format CGI pour exécuter une requête. Les servlets permettent de profiter des possibilités des navigateurs WEB pour construire l'interface graphique d'une application.

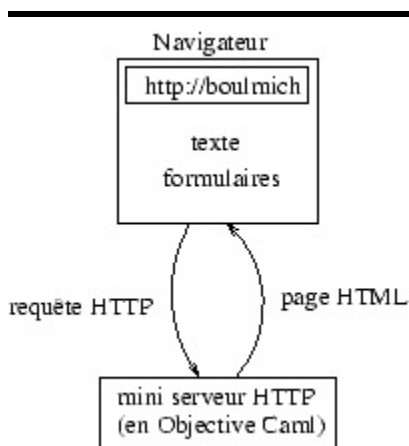


Figure 21.1 : communication entre un navigateur et un serveur Objective CAML

Nous définissons dans la suite un serveur pour le protocole HTTP. Nous ne traiterons pas l'ensemble des spécifications de ce protocole, mais nous nous limiterons aux quelques fonctions nécessaires à l'implantation d'un serveur mimant le comportement d'une application CGI.

Dans un premier temps, nous définissons un module générique de serveur Gsd. Puis nous donnons les fonctions utiles à la définition d'une application de ce serveur générique pour traiter une partie du protocole HTTP.

Formats HTTP et CGI

Nous voulons obtenir un serveur qui imite le comportement d'une application CGI. Une des premières tâches à réaliser est de décoder le format des requêtes HTTP avec les extensions CGI pour le passage des arguments.

Les clients de ce serveur pourront donc être des navigateurs tels Netscape ou Windows Explorer.

Acquisition des requêtes

Les requêtes qui transitent selon le protocole HTTP ont essentiellement trois composantes : une méthode, une URL et des données. Les données répondent à un format particulier.

Nous réalisons dans ce paragraphe un ensemble de fonctions permettant la lecture, le découpage et le décodage des composantes d'une requête. Ces fonctions pourront déclencher l'exception :

```
# exception Http_error of string ;;  
exception Http_error of string
```

Décodage

La fonction `decode`, qui utilise l'auxiliaire `rep_xcode`, a pour but de rétablir les caractères qui ont été encodés par le client HTTP : les

espaces (qui ont été remplacés par +) et certains caractères réservés qui ont été remplacés par leur code hexadécimal.

```
# let rec rep_xcode s i =  
  let xs = "0x00" in  
  String.blit s (i+1) xs 2 2;  
  String.set s i (char_of_int (int_of_string xs));  
  String.blit s (i+3) s (i+1) ((String.length s)-(i+3));  
  String.set s ((String.length s)-2) '\\000';  
  Printf.printf"rep_xcode1(%s)\\n" s ;;  
val rep_xcode : string -> int -> unit = <fun>  
  
# exception End_of_decode of string ;;  
exception End_of_decode of string  
  
# let decode s =  
  try  
    for i=0 to pred(String.length s) do  
      match s.[i] with  
      | '+' -> s.[i] <- ' '  
      | '%' -> rep_xcode s i  
      | '\\000' -> raise (End_of_decode (String.sub s 0 i))  
      | _ -> ()  
    done;  
    s  
  with  
  End_of_decode s -> s ;;  
val decode : string -> string = <fun>
```

Fonctions de manipulation de chaînes

On définit dans le module `String` plus les fonctions de découpage de chaînes de caractères :

- `prefix` et `suffix` qui extraient une sous-chaîne à partir d'un index;
- `split` qui retourne la sous-chaîne jusqu'au caractère séparateur;
- `unsplit` qui concatène deux chaînes en ajoutant un caractère

séparateur entre elles.

```
# module String_plus =
  struct
    let prefix s n =
      try String.sub s 0 n
      with Invalid_argument("String.sub") -> s

    let suffix s i =
      try String.sub s i ((String.length s)-i)
      with Invalid_argument("String.sub") -> ""

    let rec split c s =
      try
        let i = String.index s c in
        let s1, s2 = prefix s i, suffix s (i+1) in
        s1::(split c s2)
      with
        Not_found -> [s]

    let unsplit c ss =
      let f s1 s2 = match s2 with "" -> s1 | _ -> s1^(Char.es
List.fold_right f ss ""
end ;;
```

Découpage des données d'un formulaire

Les requêtes sont souvent émises depuis une page HTML contenant un formulaire. Le contenu d'un formulaire est transmis comme une chaîne de caractères contenant les noms et les valeurs associées aux champs du formulaire. La fonction `get_field_pair` transforme cette chaîne en une liste d'association.

```
# let get_field_pair s =
  match String_plus.split '=' s with
  [n;v] -> n,v
  | _ -> raise (Http_error ("Bad field format : "^s)) ;;
val get_field_pair : string -> string * string = <fun>
```



```
# let get_form_content s =
  let ss = String_plus.split '&' s in
  List.map get_field_pair ss ;;
val get_form_content : string -> (string * string) list
```

Lecture et découpage

La fonction `get_query` extrait de la requête la méthode désignée ainsi que l'URL associée qu'elle stocke dans un tableau de chaînes de caractères. On peut ainsi utiliser une application CGI qui récupère ces arguments dans le tableau des arguments de la ligne de commande. La fonction `get_query` utilise l'auxiliaire `get`. La taille maximale d'une requête est arbitrairement limitée, par nous, à 2555 caractères.

```
# let get =
  let buff_size = 2555 in
  let buff = String.create buff_size in
  (fun ic -> String.sub buff 0 (input ic buff 0 buff_size)
val get : in_channel -> string = <fun>
```

```
# let query_string http_frame =
  try
    let i0 = String.index http_frame ' ' in
    let q0 = String_plus.prefix http_frame i0 in
    match q0 with
    "GET"
    -> begin
      let i1 = succ i0 in
      let i2 = String.index_from http_frame i1 ' ' in
      let q = String.sub http_frame i1 (i2-i1) in
      try
        let i = String.index q '?' in
        let q1 = String_plus.prefix q i in
        let q = String_plus.suffix q (succ i) in
        Array.of_list (q0::q1::(String_plus.split ' ' (decode q)))
      with
      Not_found -> [|q0;q|]
    end
  end
```

```

| _ -> raise (Http_error ("Non supported method: "^q0))
with e -> raise (Http_error ("Unknown request: "^http_f
val query_string : string -> string array = <fun>

# let get_query_string ic =
  let http_frame = get ic in
  query_string http_frame;;
val get_query_string : in_channel -> string array = <fun>

```

Le serveur

Pour obtenir un pseudo-serveur CGI, qui ne sait en l'état traiter que la méthode GET, on écrit la fonction `http_service` dont l'argument `fun_serv` est une fonction de traitement des requêtes HTTP telle qu'elle aurait pu être écrite pour une application CGI.

```

# module Text_Server = Server (struct type t = string
  let to_string x = x
  let of_string x = x
end);;

# module P_Text_Server (P : PROTOCOL) =
  struct
    module Internal_Server = Server (P)

    class http_servlet  n np fun_serv =
      object(self)
        inherit [P.t] Internal_Server.server n np

        method receive_h fd =
          let ic = Unix.in_channel_of_descr fd in
          input_line ic

        method treat fd =
          let oc = Unix.out_channel_of_descr fd in (
            try
              let request = self#receive_h fd in
              let args = query_string request in
              fun_serv oc args;

```

```

with
Http_error s -> Printf.fprintf oc "HTTP error : %s <BR>" s
| _ -> Printf.fprintf oc "Unknown error <BR>" );
flush oc;
Unix.shutdown fd Unix.SHUTDOWN_ALL
end
end;;

```

Comme il n'est pas prévu de faire communiquer, via le servlet, de valeurs internes Objective CAML spéciales, on choisit le type *string* comme type du protocole. Les fonctions `of_string` et `to_string` ne font rien.

```

# module Simple_http_server =
  P_Text_Server (struct type t = string
    let of_string x = x
    let to_string x = x
    end) ;;

```

On construit alors la fonction principale de lancement du service en construisant une instance de la classe *http_servlet*.

```

# let cgi_like_server port_num fun_serv =
  let sv = new Simple_http_server.http_servlet port_num 3
  in sv#start;;
val cgi_like_server : int -> (out_channel -> string array)
    <fun>

```

Test du servlet

Il est toujours utile en cours de développement de pouvoir tester les parties déjà réalisées. Pour cela on réalise un petit serveur HTTP qui envoie tel quel le fichier inscrit dans la requête HTTP qui lui a été adressée. La fonction `simple_serv` envoie le fichier dont le nom suit la requête GET (deuxième élément du tableau des arguments). La fonction `simple_serv` trace les différents arguments passés dans la requête.

```

# let send_file oc f =
  let ic = open_in_bin f in
  try
    while true do
      output_byte oc (input_byte ic)
    done
  with End_of_file -> close_in ic;;
val send_file : out_channel -> string -> unit = <fun>

# let simple_serv oc args =
  try
    Array.iter (fun x -> print_string (x^" ")) args;
    print_newline();
    send_file oc args.(1)
  with _ -> Printf.printf "erreur\n";;
val simple_serv : out_channel -> string array -> unit =

# let run n = cgi_like_server n simple_serv;;
val run : int -> unit = <fun>

```

L'appel `run 4003` lance ce servlet sur le port 4003. Par ailleurs, on lance un navigateur effectuant la requête de chargement de la page `baro.html` sur le port 4003. La figure 21.2 montre l'affichage du contenu de cette page dans le navigateur.



Figure 21.2 : requête HTTP sur un servlet Objective CAML

Le navigateur a envoyé la requête `GET /baro.html` pour le chargement de la page, et ensuite la requête de chargement de l'image `GET /canard.gif`.

Interface HTML pour un servlet

Nous utilisons le serveur à *la CGI* pour construire une interface HTML pour la base de données du chapitre 6 (voir page ??).

Le menu de la fonction main est ici affiché sous forme d'une page HTML proposant les mêmes choix. Les réponses aux requêtes sont aussi des pages HTML dynamiquement construites par le servlet. La construction dynamique de pages fait appel à l'utilitaire que nous définissons ci-dessous.

Protocole de l'application

Nous utilisons dans notre application plusieurs éléments de plusieurs protocoles :

1. Les requêtes transitent entre un navigateur WEB et notre serveur d'application selon le format des requêtes HTTP.
2. Les données constituant les requêtes obéissent au format de codage des applications CGI.
3. Le contenu des réponses est donné selon le format des pages HTML.
4. Enfin, la nature des requêtes est donnée selon un format spécifique à cette application.

Nous voulons répondre à trois requêtes : demande de la liste des adresses postales, demande de la liste des adresses électroniques et demande de l'état des cotisations entre deux dates données. À chacune d'elles, nous associons respectivement les noms : `postal_addr`, `email_addr` et `etat_cotis`. Dans ce dernier cas, nous transmettrons également deux chaînes de caractères contenant les dates souhaitées. Ces deux dates

correspondent aux valeurs des champs debut et fin d'un formulaire HTML.

À la première connexion d'un client la page de garde suivante est envoyée. Les noms des requêtes y sont codés sous forme d'ancres HTML.

```
<HTML>
<TITLE> association </TITLE>
<BODY>
<HR>
<H1 ALIGN=CENTER> L'association</H1>
<P>
<HR>
<UL>
<LI> Liste des
<A HREF="http://freres-gras.ufr-info-p6.jussieu.fr:12345"
adresses postales
</A>
<LI> Liste des
<A HREF="http://freres-gras.ufr-info-p6.jussieu.fr:12345"
adresses électroniques
</A>
<LI> État des cotisations<BR>
<FORM
  method="GET"
  action="http://freres-gras.ufr-info-p6.jussieu.fr:12345"
Date de début : <INPUT type="text" name="debut" value=""
Date de fin : <INPUT type="text" name="fin" value=""
<INPUT name="action" type="submit" value="Envoyer">
</FORM>
</UL>
<HR>
</BODY>
</HTML>
```

Nous supposerons que cette page est contenue dans le fichier assoc.html.

Primitives pour HTML

Les fonctionnalités de l'utilitaire HTML sont réunies au sein de la seule classe `print`. Elle possède un champ indiquant le canal de sortie. Elle peut donc aussi bien être utilisée dans le cadre d'une application CGI (où le canal de sortie est la sortie standard) que d'une application utilisant le serveur HTTP défini au paragraphe précédent (où le canal de sortie est une socket de service).

Les méthodes proposées permettent essentiellement d'encapsuler du texte dans des balises HTML. Ce texte est, soit passé directement en argument aux méthodes sous forme de chaîne de caractères, soit produit par une fonction. Par exemple, la méthode principale `page` prend en premier argument une chaîne correspondant à l'en-tête de la page¹, et en second argument une fonction affichant le contenu de la page. La méthode `page` produit les balises correspondantes du protocole HTML.

Le nom des méthodes reprend le nom des balises HTML correspondantes en y ajoutant parfois quelques options.

```
# class print (oc0:out_channel) =  
  object (self)  
    val oc = oc0  
    method flush () = flush oc  
    method str =  
      Printf.fprintf oc "%s"  
    method page header (body:unit -> unit) =  
      Printf.fprintf oc "<HTML><HEAD><TITLE>%s</TITLE></HEAD>  
body() ;  
      Printf.fprintf oc "</BODY>\n</HTML>\n"  
    method p () =  
      Printf.fprintf oc "\n<P>\n"  
    method br () =  
      Printf.fprintf oc "<BR>\n"  
    method hr () =  
      Printf.fprintf oc "<HR>\n"  
    method hr () =  
      Printf.fprintf oc "\n<HR>\n"  
    method h i s =  
      Printf.fprintf oc "<H%d>%s</H%d>" i s i  
    method h_center i s =
```

```

Printf.fprintf oc "<H%d ALIGN=\"CENTER\">%s</H%d>" i s
method form url (form_content:unit -> unit) =
Printf.fprintf oc "<FORM method=\"post\" action=\"%s\">
form_content ();
Printf.fprintf oc "</FORM>"
method input_text =
Printf.fprintf oc
"<INPUT type=\"text\" name=\"%s\" size=\"%d\" value=\"%s\">"
method input_hidden_text =
Printf.fprintf oc "<INPUT type=\"hidden\" name=\"%s\" value=\"%s\">"
method input_submit =
Printf.fprintf oc "<INPUT name=\"%s\" type=\"submit\" value=\"%s\">"
method input_radio =
Printf.fprintf oc "<INPUT type=\"radio\" name=\"%s\" value=\"%s\">"
method input_radio_checked =
Printf.fprintf oc
"<INPUT type=\"radio\" name=\"%s\" value=\"%s\" CHECKED\">"
method option =
Printf.fprintf oc "<OPTION> %s\n"
method option_selected opt =
Printf.fprintf oc "<OPTION SELECTED> %s" opt
method select name options selected =
Printf.fprintf oc "<SELECT name=\"%s\">\n" name;
List.iter
(fun s -> if s=selected then self#option_selected s else
options;
Printf.fprintf oc "</SELECT>\n"
method options selected =
List.iter
(fun s -> if s=selected then self#option_selected s else
end ;;

```

Nous supposerons que cet utilitaire est fourni par le module `Html_frame`.

Pages dynamiques pour la gestion d'associations

Pour chacune des trois requêtes de l'application, il faut construire

une page en réponse. Nous utilisons pour cela l'utilitaire `Html_frame` donné ci-dessus. Ce qui signifie que les pages ne sont pas réellement construites, mais que leurs différents composants sont émis séquentiellement sur le canal de sortie.

Nous ajoutons une page (virtuelle) supplémentaire qui est retournée en réponse à une requête erronée ou incomprise.

Page d'erreur

La fonction `print_error` prend en argument une fonction d'émission de page HTML (i.e. : une instance de la classe `print`) et une chaîne de caractères contenant le message d'erreur.

```
# let print_error (print:Html_frame.print) s =  
  let print_body() =  
    print#str s; print#br()  
  in  
    print#page "Erreur" print_body ;;  
val print_error : Html_frame.print -> string -> unit = <
```

Toutes nos fonctions d'émission d'une réponse à une requête auront en paramètre un premier argument contenant la fonction d'émission d'une page HTML.

Liste des adresses postales

La page composant la réponse à la demande de la liste des adresses postales est obtenue en formatant la liste des chaînes de caractères obtenue par la fonction `adresses_postales` définie pour la base d'adhérents (voir page [??](#)). Nous supposons que cette fonction, et toute autre concernant directement les requêtes sur la base de données, ont été définies dans un module nommé `Assoc`.

Pour émettre cette liste, nous utilisons une fonction de sortie de lignes simples :

```
# let print_lines (print:Html_frame.print) ls =
```

```

let print_line l = print#str l; print#br() in
List.iter print_line ls ;;
val print_lines : Html_frame.print -> string list -> unit

```

La fonction de réponse à la demande des adresses postales est :

```

# let print_adresses_postales print db =
  print#page "Adresses postales"
  (fun () -> print_lines print (Assoc.adresses_postales db))
;;
val print_adresses_postales : Html_frame.print -> Assoc.data_base -> unit = <fun>

```

Outre la fonction d'émission de page, la fonction `print_adresses_postales` prend en second paramètre la base de données.

Liste des adresses électroniques

Cette fonction est construite sur le même principe que celle donnant la liste des adresses postales sauf qu'elle fait appel à la fonction `adresses_electroniques` du module `Assoc` :

```

# let print_adresses_electroniques print db =
  print#page "Adresses électroniques"
  (fun () -> print_lines print (Assoc.adresses_electroniques db))
;;
val print_adresses_electroniques :
  Html_frame.print -> Assoc.data_base -> unit = <fun>

```

État des cotisations

C'est encore le même principe qui gouverne la définition de cette fonction : récupérer les données correspondant à la requête (qui ici est un couple), puis émettre les chaînes de caractères correspondantes.

```

# let print_etat_cotisations print db d1 d2 =
  let ls, t = Assoc.etat_cotisations db d1 d2 in

```

```

let page_body() =
  print_lines print ls;
  print#str ("Total : "^(string_of_float t));
  print#br()
in
  print#page "État des cotisations" page_body ;;
val print_etat_cotisations :
  Html_frame.print -> Assoc.data_base -> string -> string

```

Analyse des requêtes et réponse

Nous définissons deux fonctions de production de réponse en fonction d'une requête HTTP. La première (`print_get_answer`) répond à une requête supposée formulée par une méthode GET du protocole HTTP. La seconde aiguille la production de la réponse selon la méthode de requête utilisée.

Ces deux fonctions prennent en second argument un tableau de chaînes de caractères contenant les éléments de la requête HTTP tels qu'ils ont été analysés par la fonction `get_query_string` (voir page ??). Le premier élément du tableau contient la méthode et le second le nom de la requête sur la base.

Dans le cas d'une demande d'état des cotisations, les dates de début et de fin composant la requête sont contenues dans les deux champs du formulaire associé à cette demande. Les données du formulaire sont contenues dans le troisième champ du tableau qui doit être décomposé par la fonction `get_form_content` (voir page ??).

```

# let print_get_answer print q db =
  match q.(1) with
  | "/postal_addr" -> print_adresses_postales print db
  | "/email_addr" -> print_adresses_electroniques print db
  | "/etat_cotis"
  -> let nvs = get_form_content q.(2) in
    let d1 = List.assoc "debut" nvs
    and d2 = List.assoc "fin" nvs in
      print_etat_cotisations print db d1 d2

```

```

| _ -> print_error print ("Unknown request: "^q.(1)) ;;
val print_get_answer :
  Html_frame.print -> string array -> Assoc.data_base ->

# let print_answer print q db =
  try
    match q.(0) with
    "GET" -> print_get_answer print q db
    | _ -> print_error print ("Unsupported method : "^q.(0))
  with
  e
  -> let s = Array.fold_right (^) q "" in
    print_error print ("Some thing wrong with request: "^s)
val print_answer :
  Html_frame.print -> string array -> Assoc.data_base ->

```

Programme principal et application

Le programme principal de l'application est un exécutable autonome paramétré par le numéro de port du service. La base de données est lue avant le lancement du serveur. La fonction de service est obtenue à partir de la fonction `print_answer` définie ci-dessus et de la fonction générique de serveur HTTP `cgi_like_server` définie au paragraphe précédent (voir page ??). Cette dernière est donnée par le module `Servlet`.

```

# let get_port_num() =
  if (Array.length Sys.argv) < 2 then 12345
  else
    try int_of_string Sys.argv.(1)
    with _ -> 12345 ;;
val get_port_num : unit -> int = <fun>

# let main() =
  let db = Assoc.read_base "assoc.dat" in
  let assoc_answer oc q = print_answer (new Html_frame.pr
    Servlet.cgi_like_server (get_port_num()) assoc_answer ;

```

```
val main : unit -> unit = <fun>
```

Pour obtenir l'application complète, nous rassemblons dans un fichier `httpassoc.ml` les définitions des fonctions d'affichage. Ce fichier se termine par un appel à la fonction `main` :

```
main() ;;
```

Nous pouvons alors produire un exécutable nommé `assocd` par la commande de compilation :

```
ocamlc -thread -custom -o assocd unix.cma threads.cma \
      gsd.cmo servlet.cmo html_frame.cmo string_plus.cm
      httpassoc.ml -cclib -lunix -cclib -lthreads
```

Ne reste plus alors qu'à lancer le serveur, charger la page HTML² contenue dans le fichier `assoc.html` donné au début de ce paragraphe (page ??) et cliquer.

La figure 21.3 montre un exemple d'utilisation.

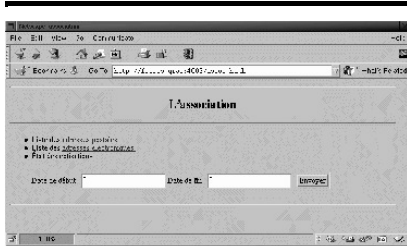


Figure 21.3 : requête HTTP sur un servlet Objective CAML

Le navigateur effectue une première connexion sur le servlet qui lui envoie la page de menu. Une fois les champs de saisie remplis, l'utilisateur envoie une nouvelle requête qui contient les champs saisis. Ceux-ci sont décodés, et le serveur fait un accès à la base de données de l'association pour récupérer l'information demandée qui est traduite en HTML, envoyée au client qui affiche cette nouvelle

page.

Pour en faire plus

Cette application a de nombreux prolongements. Tout d'abord le protocole HTTP utilisé est trop simple par rapport aux nouvelles versions qui ajoutent un en-tête informatif sur le type, le contenu et la longueur de la page envoyée. De même la méthode POST, qui autorise une modification sur le serveur, n'est pas intégrée³ Pour pouvoir décrire le type et le contenu des pages renvoyées, il est nécessaire d'intégrer dans les servlets la convention MIME utilisée pour la description des documents comme pour les documents attachés dans les courriers électroniques.

La transmission d'images, comme à la figure 21.2, permet aussi de construire des interfaces pour les jeux à 2 joueurs (voir chapitre 17), où l'on associe des liens aux dessins des cases à jouer. Comme l'arbitre du jeu connaît les coups légaux, seules les cases valides sont associées à un lien.

L'extension MIME autorise aussi à définir de nouveaux types de données. On intègre alors le protocole interne des valeurs Objective CAML comme un nouveau type MIME. Ces valeurs ne sont réellement compréhensibles que par un programme Objective CAML utilisant le même protocole interne. Ainsi une requête d'un client sur une valeur Objective CAML distante est effectuée via une requête HTTP. On peut ainsi passer dans la page HTML une fermeture sérialisée qui sera passée en tant qu'argument de la requête. Celle-ci, une fois reconstruite du côté du serveur s'exécute pour fournir le résultat escompté.

Développement d'applications en Objective CAML

Introduction

À cette étape de sa lecture, le lecteur ne doit plus douter de la

richesse d'Objective CAML. Ce langage repose sur un noyau fonctionnel et impératif et il intègre les deux grands modèles d'organisation d'applications que sont les modules et les objets. Bien que présentés comme des bibliothèques, les processus légers sont partie prenante du langage. Les primitives système, en majeure partie portables, complètent le langage avec toutes les possibilités offertes par la programmation répartie. Ces différents paradigmes de programmation se moulent dans le cadre général du typage statique avec inférence. Pour autant, ces éléments ne tranchent pas, à eux seuls, la question de sa pertinence pour le développement d'applications, ou plus prosaïquement : « est-ce un bon langage ? ».

On ne peut à son propos utiliser l'un des arguments classiques suivants :

- (tendance marketing) « c'est un bon langage car les clients l'achètent » ;
- (tendance historique) < < c'est un bon langage car il existe des milliers de lignes déjà écrites avec > > ;
- (tendance système) < < c'est un bon langage car les systèmes Unix ou Windows sont écrits avec > > ;
- (tendance application phare) < < c'est un bon langage car telle ou telle application a été écrite avec > > ;
- (tendance normalisation) < < c'est un bon langage car il possède une spécification ISO > > .

Nous allons passer en revue une dernière fois les divers traits du langage mais cette fois sous l'angle de la pertinence de la réponse qu'il peut apporter aux besoins d'une équipe de développement. Les critères retenus pour formuler nos éléments d'évaluation prennent en compte les qualités intrinsèques du langage, son environnement de développement, les contributions de la communauté et les applications significatives réalisées. Nous confrontons enfin Objective CAML avec plusieurs langages fonctionnels proches ainsi que le langage à objets Java pour en souligner les principales différences.

Éléments d'évaluation

La distribution d'Objective CAML fournit deux compilateurs en

ligne, un engendrant du code-octet et l'autre produisant des instructions pour les processeurs les plus courants. La boucle d'interaction utilise le compilateur de code-octet. En outre, la distribution offre de nombreuses bibliothèques sous forme de modules et quelques outils pour le calcul de dépendances entre modules, le profilage et la mise au point. Enfin grâce à son interface avec le langage C, il est possible de lier des programmes Objective CAML et des programmes C. Les langages offrant également une interface avec C, comme le JNI (Java Native Interface) de Java deviennent ainsi accessibles.

Le manuel de référence donne la syntaxe du langage, décrit les outils de développement et les signatures des bibliothèques. Cet ensemble (langage, outils, bibliothèques, documentation) forment un environnement de développement.

Langage

Spécification et implantation

Il y a deux manières d'aborder un nouveau langage. Une première est de lire la spécification du langage pour avoir une vision globale. Une deuxième est de se plonger dans le guide d'utilisateur du langage en suivant les concepts illustrés par des exemples. Objective CAML ne possède ni l'un ni l'autre, ce qui rend son abord de manière autonome relativement difficile. L'absence de spécification formelle (comme en possède SML) ou descriptive (comme pour ADA) est un handicap pour comprendre le fonctionnement d'un programme. Une autre conséquence de l'absence de spécification est l'impossibilité d'obtenir un label de normalisation du langage (ISO, ANSI ou IEEE). Cela limite fortement la construction de nouvelles implantations munies d'autres environnements. Heureusement l'implantation de l'INRIA est de bonne qualité et surtout les sources de la distribution sont disponibles.

Syntaxe

Les particularités syntaxiques sont une difficulté non négligeable

pour aborder Objective CAML. Elles s'expliquent par l'origine fonctionnelle du langage, mais aussi par des facteurs historiques, voire anecdotiques.

La syntaxe de l'application d'une fonction à ses arguments est définie par la simple juxtaposition, comme dans *f 1 2*. L'absence de parenthèse perturbe aussi bien le néophyte que le programmeur C ou le programmeur Lisp confirmé. Néanmoins, cette difficulté ne vaut vraiment que pour la lecture du code d'un programmeur chiche en parenthèses. Rien n'empêche le programmeur néophyte en Objective CAML d'utiliser un parenthésage plus explicite et d'écrire (*f 1 2*).

Au delà du noyau fonctionnel, Objective CAML adopte une syntaxe rompant parfois avec l'usage : l'accès aux éléments d'un tableau utilise la notation *t.(i)* et non les usuels crochets ; l'invocation de méthode se note avec un dièse (caractère #) et non le point, etc. Ce particularisme ne facilite pas la prise en main d'Objective CAML.

Enfin, les syntaxes d'Objective CAML et de son ancêtre Caml ont subi de nombreuses modifications depuis leurs premières implantations. Ce qui n'a pas plaidé en faveur de la pérennité des applications développées.

Pour ne pas rester sur une note négative, la syntaxe de filtrage de motifs, héritée de la famille ML, qu'intègre Objective CAML structure agréablement et avec simplicité les définitions de fonctions par cas.

Typage statique

La caractéristique fondamentale du langage Objective CAML demeure le typage statique des expressions et déclarations du langage. Il garantit qu'aucune erreur de type ne survient durant l'exécution d'un programme. L'inférence statique de type a été conçue pour les langages fonctionnels de la famille ML, et Objective CAML a su maintenir la plus grande partie de cette discipline de typage pour les extensions impérative et objet. Cependant, dans le cas de l'emploi d'objets, le programmeur doit parfois épauler l'inférence de types par des contraintes explicites de typage. Même

alors, Objective CAML conserve un typage statique des expressions et des définitions, ce qui constitue un gage incontesté de sûreté d'exécution : un programme Objective CAML ne retournera pas l'exception « méthode non trouvée », ce qui n'est pas le cas des langages à objets dynamiquement typés.

Le polymorphisme paramétrique des types d'Objective CAML permet l'implantation d'algorithmes généraux. Il est reconduit dans la couche objet où les classes paramétrées produisent un code générique, et non une expansion de code comme en engendrent les templates d'autres langages. À lui seul, le polymorphisme paramétrique est une composante importante de la réutilisabilité de code.

L'extension objet ajoute une notion de polymorphisme d'inclusion qui est obtenu en spécifiant les relations de sous-typage entre objets. Il concilie la réutilisabilité de code, qui fait la force de la relation d'héritage entre classes, avec la sûreté du typage statique.

Bibliothèques et outils

Les bibliothèques fournies avec la distribution couvrent de larges besoins. Le programmeur y retrouve, en standard, l'implantation des structures de données les plus usuelles avec leurs fonctions de base. Citons : piles, files d'attente, tables de hachage, AVL. On y trouve également des outils plus avancés comme le traitement des flots de données. Ces bibliothèques s'enrichissent au fil des versions du langage.

La bibliothèque Unix autorise une programmation de plus bas niveau tant pour les entrées-sorties, que pour la gestion des processus. Elle n'est pas identique pour toutes les plate-formes ce qui limite en partie son utilisation.

La bibliothèque d'arithmétique exacte et la bibliothèque d'expressions régulières facilitent le développement d'applications spécifiques.

On peut regretter que la bibliothèque graphique portable offre peu de fonctionnalités permettant la construction d'interfaces

graphiques.

Des bibliothèques C permettent un interfaçage aisé entre ce langage et Objective CAML. Ici, la libre disponibilité de sources bien structurées et dûment commentées est un facteur d'ouverture certain vers l'extérieur et les bibliothèques diverses et variées que l'on peut y trouver.

Parmi les outils fournis, on peut citer plus spécialement ceux d'analyses lexicale et syntaxique indispensables lorsque l'on doit traiter des données textuelles complexes. Basés sur les classiques lex et yacc, ils s'intègrent parfaitement aux types somme et à la fonctionnalité d'Objective CAML, les rendant ainsi plus simples d'utilisation que leurs aînés.

Enfin, quelle que soit la solidité qu'apportent ses traits, l'utilisation d'un langage « en condition réelle » n'évite jamais les phases de mise au point d'un programme. La distribution Objective CAML 2.04 ne fournit pas d'environnement intégré de développement. Certes, l'utilisation de la boucle d'interaction permet de procéder rapidement à la compilation et aux tests unitaires des fonctions (ce qui est un avantage indéniable), mais cette utilisation variera selon les systèmes et les programmeurs : copier-coller sous X-Windows, appel de l'interprète de commande sous emacs, demande d'évaluation d'un tampon sous Windows. La version suivante (voir annexe B) propose un premier environnement pour Unix contenant un navigateur d'interfaces de modules et un éditeur structuré lié à la boucle d'interaction. Enfin le debugger de la distribution reste d'emploi difficile (en particulier, à cause de l'aspect fonctionnel et du polymorphisme paramétrique du langage) et limité au système Unix.

Documentation

La documentation de la distribution est constituée par le manuel de référence en version imprimable (PostScript) ou en ligne (HTML). Ce manuel n'est en aucun cas une introduction au langage. Par contre il est indispensable pour connaître la teneur des bibliothèques ou les paramètres des commandes. On trouve du matériel pédagogique à la page de Caml de l'Inria, mais il porte

principalement sur le langage Caml-Light. Il manque donc un manuel d'apprentissage complet du langage, nous espérons que cet ouvrage comble ce manque.

Autres outils de développement

Nous nous sommes cantonnés jusqu'à maintenant à la distribution d'Objective CAML. Néanmoins la communauté des développeurs dans ce langage est active, comme l'indique le nombre de transactions de la liste de diffusion `caml-list@inria.fr`. De nombreux outils, bibliothèques et extensions sont utilisés pour faciliter le développement. On détaille par la suite l'utilisation d'outils d'édition, d'extension de syntaxe, d'interfaçage avec d'autres langages et de programmation parallèle. Nous mentionnons également les nombreuses bibliothèques d'interface graphique. On retrouve la plupart de ces contributions sur le site de la < < bosse du chameau > > .

Outils d'édition

Il existe plusieurs modes reconnaissant la syntaxe Objective CAML pour l'éditeur emacs. L'intérêt de ces modes est d'indenter automatiquement le texte en cours d'écriture, permettant ainsi une meilleure lisibilité de celui-ci. C'est une alternative à la fenêtre d'interaction sous Windows. Puisqu'emacs fonctionne sous Windows, on peut lancer dans une de ses fenêtres la boucle d'interaction Objective CAML.

Extension de syntaxe

Les outils d'analyses lexicale et syntaxique proposés dans la distribution sont déjà fort complets, mais ils ne permettent pas d'étendre la syntaxe du langage lui-même. L'outil `camlp4` (voir lien page ??) s'utilise en lieu et place de l'analyseur syntaxique d'Objective CAML. Les compilateurs de ce dernier n'ont plus qu'à procéder au typage et à la génération de code. Cet outil permet l'extension de la syntaxe du langage par l'utilisateur ou la modification de la syntaxe originale du langage. Il offre également

des fonctionnalités d'affichage (pretty-printing) des arbres de syntaxe engendrés. Il devient alors facile d'écrire une nouvelle boucle d'interaction pour toute extension de la syntaxe d'Objective CAML, voire pour un autre langage implanté en Objective CAML.

Interopérabilité avec d'autres langages

Le chapitre [12](#) a détaillé l'interfaçage entre le langage Objective CAML et C. Une application multi-langages tire avantage des traits utiles de chacun d'eux tout en faisant vivre en harmonie des codes différents partageant un même espace mémoire. Néanmoins l'encapsulation de fonctions C pour les rendre appelables à partir d'Objective CAML nécessite un travail fastidieux. Pour le simplifier, l'outil `camlIDL` (voir lien page [??](#)) fournit un générateur de capsules et les outils d'importation de composants COM (Windows) en Objective CAML. Les capsules sont engendrées à partir d'un fichier des descriptions de l'interface (IDL).

Interfaces graphiques

La bibliothèque Graphics permet de réaliser des dessins et des interactions simples, mais elle ne peut pas être considérée comme une interface graphique digne de ce nom. Le chapitre [13](#) a montré comment on pouvait étendre cette bibliothèque pour construire des composants graphiques répondant à certaines interactions. Le fait de reposer sur Graphics permet de conserver la portabilité de l'interface sur différentes plate-formes (X-Windows, Windows, MacOS), mais limite son utilisation à la couche basse des événements et des contextes graphiques.

Plusieurs réalisations tentent de combler ce manque, malheureusement aucune n'arrive à être complète, portable, documentée et simple d'emploi. Voici une liste (extraite de la « bosse du chameau ») des principales réalisations :

- OlibRt : sous ce doux nom, c'est une véritable boîte à outils, construite au dessus de X-Windows, mais non documentée. Sa distribution contient des exemples complets en particulier de nombreux jeux.

Lien: <http://cristal.inria.fr/~ddr/>

- camlTk est une interface complète du toolkit Tk bien documentée. Son point faible est de dépendre des versions de Tcl/Tk ce qui la rend difficile à installer. Elle a été utilisée pour la construction du navigateur mmm [Rou96] écrit en Objective CAML.

Lien: <http://caml.inria.fr/~rouaix/camltk-readme.html>

- La bibliothèque Xlib a été réécrite en Objective CAML. Efun, mini-clone d'emacs est réalisé en l'utilisant. Xlib n'est pas vraiment une boîte à outils, et n'est pas portable sur d'autres systèmes graphiques que X-Windows.
- mlGtk est une interface construite sur Gtk. Elle est en cours de réalisation et n'a pas de documentation. Son intérêt est d'être portable sous Unix et Windows (car Gtk l'est) de manière plus simple que Tk. D'autre part elle utilise la couche objet d'Objective CAML, ce qui n'est pas sans poser parfois quelques problèmes.
- LabTk est une interface avec Tcl/Tk, pour Unix, utilisant des extensions d'Objective CAML qui seront intégrées dans la prochaine version (voir annexe B). Elle contient sa propre distribution de Tcl/Tk qui s'installe facilement.

Malgré l'effort de la communauté, il existe un réel manque d'outils de construction d'interfaces portables. On peut espérer que LabTk devienne portable sur différents systèmes.

Programmation parallèle et distribution

Les processus légers et les prises de communication offrent déjà les mécanismes de base pour la programmation concurrente et répartie. L'interfaçage avec le langage C permet d'utiliser les bibliothèques classiques de programmation parallèle. Seule manque une interface avec CORBA pour l'invocation de méthodes sur des objets distants. Par contre, il existe de nombreuses bibliothèques et extensions du langage qui utilisent différents modèles de parallélisme.

Bibliothèques

Les deux principales bibliothèques pour la programmation parallèle, MPI (Message Passing Interface) et PVM (Parallel Virtual Machine), sont interfacées avec Objective CAML. On trouve sur le site

Lien: <http://www.netlib.org>

la documentation, les liens et les sources de ces bibliothèques. La « bosse du chameau » contient les différentes adresses HTTP d'où l'on peut télécharger les versions interfacées avec Objective CAML.

Extensions

De nombreuses extensions parallèles de Caml-Light ou Objective CAML ont été réalisées :

- Caml-Flight ([[FC95](#)]) est une extension SPMD (Simple Program Multiple Data) du langage Caml-Light. Un programme exécute une copie de lui-même sur un nombre fixé de processus. Les communications sont explicites, il existe seulement une opération de communication get qui peut être exécutée uniquement à l'intérieur de l'opération de synchronisation sync.

Lien: <http://www.univ-orleans.fr/SCIENCES/LIFO/Members/ghains/caml-flight.html>

- BSML [[Lou98](#)] est une extension par des opérations BSP. Le langage préserve la compositionnalité et permet des prévisions précises de performances si le nombre de processeurs est fixé.

Lien: <http://www.univ-orleans.fr/SCIENCES/LIFO/Members/loulergu/bsml.html>

- OCAMLP3 [[DDL98](#)] est un environnement de programmation parallèle basé sur le modèle des squelettes du langage P3L. Les différents squelettes prédéfinis peuvent s'imbriquer. Les programmes peuvent se tester soit en mode

séquentiel soit en mode parallèle permettant ainsi de réutiliser des outils propres à Objective CAML.

Lien: <http://www.di.unipi.it/~susanna/projects.html>

- JoCAML [[CL99](#)] repose sur le modèle du join-calculus qui permet des opérations de haut niveau pour la concurrence, la communication et la synchronisation en présence d'objets répartis et de codes mobiles tout en conservant une gestion automatique de la mémoire.

Lien: <http://pauillac.inria.fr/jocaml/>

- Lucid Synchrone ([[CP95](#)]) est un langage dédié à l'implantation de systèmes réactifs. Il combine la fonctionnalité d'Objective CAML et les traits des langages de flots synchrones.

Lien: <http://www-spi.lip6.fr/~pouzet/lucid-synchrone/>

Applications réalisées en Objective CAML

Un certain nombre d'applications ont été réalisées en Objective CAML. Nous ne parlerons que des applications « publiques », c'est-à-dire que l'on peut utiliser soit gratuitement soit en les achetant.

Comme les autres langages fonctionnels, Objective CAML est un bon langage d'implantation de compilateurs. Le bootstrap¹ du compilateur ocaml en est un exemple probant. De même de nombreuses extensions au langage ont été apportées, comme vu précédemment pour la programmation parallèle, mais aussi sur le typage comme O'Labl (dont une partie est en cours d'intégration en Objective CAML, voir annexe [B](#)) ou sur les unités physiques. Les liens vers ces applications se retrouvent sur la « bosse du chameau ».

La deuxième spécialité d'Objective CAML concerne les systèmes d'aide à la preuve. Le développement majeur en ce domaine est le logiciel Coq qui accompagne quasiment depuis son origine

l'évolution de Caml. Historiquement, ML a été conçu comme langage de manipulation du système LCF (Logic for Computable Functions), avant de devenir indépendant de cette application. Il est donc naturel de le retrouver comme langage d'implantation d'un important système d'aide à la preuve.

Un troisième domaine d'application concerne le parallélisme (voir page ??) et la communication dont un bon exemple est le système Ensemble.

Lien: <http://www.cs.cornell.edu/Info/Projects/Ensemble/>

Une liste, non exhaustive, d'applications significatives développées en Objective CAML est maintenue sur le site Caml de l'Inria :

Lien: http://caml.inria.fr/users_programs-eng.html

Citons en particulier hevea qui est un traducteur L^AT_EX vers HTML que nous avons utilisé pour réaliser la version HTML de ce livre qui se trouve sur le cédérom l'accompagnant.

Lien: <http://pauillac.inria.fr/~maranget/hevea/>

Quoique d'importance, les applications que nous venons de citer ne représentent pas ce, qu'en début de ce chapitre, nous avons baptisé une « application phare ». De même, elles n'explorent pas un nouveau domaine de spécialité montrant la pertinence d'utilisation d'Objective CAML. Il n'est pas évident que cet exemple puisse être issu du monde académique. Il est plus probable qu'il vienne du monde industriel, soit en liaison avec une normalisation du langage (voire sa spécification formelle), soit pour les besoins d'applications devant intégrer divers styles de programmation et de structuration du logiciel.

Langages fonctionnels proches

Il existe plusieurs langages proches d'Objective CAML, soit par le côté fonctionnel, soit par le typage. Objective CAML est issu de la famille ML, et possède donc des cousins dont les plus proches sont outre-atlantique et outre-manche dans la lignée de SML (Standard ML). La famille Lisp, et en particulier le langage Scheme, diffère de

ML principalement par son typage dynamique. Deux langages paresseux, Miranda et Haskell, reprennent ou étendent le typage de ML dans le cadre de l'évaluation retardée. Deux langages fonctionnels Erlang et SCOL développés respectivement par les sociétés Ericsson et Cryo-Networks sont tournés vers la communication.

Famille ML

La famille ML comprend deux branches principales : Caml (Categorical Abstract Machine Language) et ses dérivés Caml-Light et Objective CAML, SML (Standard ML) et ses descendants SML/NJ et mossm. Caml, l'ancêtre, a été développé entre 1986 et 1990 par le projet FORMEL de l'INRIA en collaboration avec l'Université Paris 7 et l'École Normale Supérieure. Son implantation reposait sur la bibliothèque d'exécution de `Le_Lisp`. Il intégrait dans le langage la définition de grammaires et d'afficheurs ce qui permet la communication de valeurs entre le langage décrit et Caml. Son système de types était plus contraignant pour les valeurs physiquement modifiables en ce qu'il n'autorisait pas que de telles valeurs fussent polymorphes. Son premier descendant, Caml-Light, n'utilisait plus la machine CAM, mais la Zinc pour l'implantation. Le nom a néanmoins été conservé pour montrer sa filiation. Il a apporté une implantation plus légère en optimisant l'allocation des fermetures et utilisait un GC performant précurseur du GC actuel. Cet allègement permettait de l'utiliser sur les micros-ordinateurs de l'époque. Les différentes versions de Caml-Light ont évolué vers le typage actuel des traits impératifs et se sont enrichies de nombreuses bibliothèques. Le rejeton suivant, Caml Special Light ou CSL, a introduit les modules paramétrés et le compilateur natif. Enfin le benjamin est actuellement Objective CAML qui ajoute principalement l'extension objet à CSL. Comme il n'y a jamais eu de spécification complète du langage Caml, ces différentes évolutions ont pu s'effectuer en toute liberté.

L'approche SML a été inverse. La spécification formelle [[MTH90](#)] a été donnée avant la première implantation. Elle est difficile à lire, et un deuxième livre en donne un commentaire ([[MT91](#)]). Cette démarche, spécification puis implantation, a permis la réalisation de

plusieurs implantations, dont la plus connue est SML/NJ (Standard ML of New Jersey) de Lucent (ex-ATT). Dès l'origine, SML a intégré des modules paramétrés. Son système de typage initial était différent de celui de Caml pour les traits impératifs en introduisant un niveau de faiblesse sur les variables de type. Les différences entre les deux langages sont détaillées dans [CKL96]. Ces différences s'estompent avec le temps. Les deux familles ont le même système de type pour le noyau fonctionnel et impératif, Objective CAML possède maintenant des modules paramétrés. SML a aussi subi des évolutions, le rapprochant d'Objective CAML comme pour les types enregistrements. Si les deux langages ne fusionnent pas, cela provient principalement de leur développement séparé. Il est à noter qu'il existe un environnement de développement commercial pour SML, MLWorks, de chez Harlequin :

Lien: <http://www.harlequin.com/products/>

Une implantation de SML, `mossm`, reposant sur la bibliothèque d'exécution de Caml-Light a aussi été implantée.

Scheme

Le langage Scheme (1975) est un dialecte du langage Lisp (1960). Il est normalisé (IEEE Std 1178-1990). C'est un langage fonctionnel à évaluation stricte, muni de traits impératifs, typé dynamiquement. Sa syntaxe est régulière et particulière par l'usage des parenthèses. La structure de donnée principale est la paire pointée (équivalent du couple ML) avec laquelle on construit des listes possiblement hétérogènes. La boucle principale d'une boucle d'interaction Scheme s'écrit (`print (eval (read))`). La fonction `read` lit l'entrée standard et construit une expression Scheme. La fonction `eval` évalue l'expression construite et la fonction `print` affiche le résultat. Scheme possède un système de macro-expansion très pratique qui, associé à la fonction `eval`, permet de construire facilement des extensions du langage. Il permet non seulement d'effectuer des ruptures de calcul (exceptions) mais aussi des reprises de calcul grâce aux continuations. Une continuation correspond à un point de calcul. La forme spéciale `call_cc` lance un calcul avec la possibilité

de reprendre celui-ci au niveau de la continuation courante, c'est à dire du retour de ce calcul. Il existe de très nombreuses implantation de Scheme. Il est même utilisé comme langage de macros pour le logiciel de retouche d'images GIMP. Scheme est un excellent laboratoire expérimental pour l'implantation de nouveaux concepts de programmation séquentielle ou parallèle (grâce aux continuations).

Langages à évaluation retardée

À la différence de ML ou Lisp, les langages à évaluation retardée ne calculent pas les paramètres d'appel des fonctions à leur passage, mais quand l'évaluation du corps de la fonction le nécessite. Il existe une version « paresseuse » de ML appelée Lazy ML mais les représentants principaux de cette famille de langages sont Miranda et Haskell.

Miranda

Miranda([[Tur85](#)]) est un langage fonctionnel pur. C'est à dire sans effet de bord. Un programme Miranda est une suite d'équations définissant les fonctions et les structures de données.

Lien: <http://www.engin.umd.umich.edu/CIS/course.des/cis400/miranda/miranda.html>

Par exemple la fonction fib se définit ainsi :

```
fib a = 1, a=0
      = 1, a=1
      = fib(a-1) + fib(a-2), a>1
```

La sélection des équations se fait soit par des gardes (expressions conditionnelles) comme ci-dessus, soit par un filtrage de motif comme dans l'exemple ci-dessous :

```
fib 0 = 1
fib 1 = 1
fib a = fib(a-1) + fib(a-2)
```

Ces deux méthodes peuvent se mélanger.

Les fonctions sont d'ordre supérieur et peuvent être évaluées partiellement. L'évaluation est paresseuse, aucune sous-expression n'est calculée jusqu'au moment où sa valeur devient nécessaire. Ainsi, les listes Miranda sont naturellement des flots.

Miranda a une syntaxe concise pour les structures infinies (listes, ensembles) : `[1..]` représente la liste de tous les entiers. La liste des valeurs de la fonction de Fibonacci s'écrit brièvement : `fibs = [a | (a,b) <- (1,1), (b,a+b) ..]`. Comme les valeurs ne sont calculées que pour leur utilisation, la déclaration de `fibs` ne coûte rien.

Miranda est fortement typé en utilisant un système de type à la Hindley-Milner. Sa discipline de type est essentiellement la même que ML. Il accepte la définition de données par l'utilisateur.

Miranda est l'archétype des langages fonctionnels purs paresseux.

Haskell

Le site principal du langage Haskell contient les rapports de définition du langage et de ses bibliothèques, ainsi que les principales implantations de celui-ci.

Lien: <http://www.haskell.org>

Plusieurs ouvrages sont consacrés à la programmation fonctionnelle en Haskell, un des plus récents est [[Tho99](#)]. C'est un langage qui reprend presque tous les nouveaux concepts des langages fonctionnels. Il est pur (sans effet de bord), paresseux (non-strict), muni d'un polymorphisme *ad hoc* (pour la surcharge) en plus du polymorphisme paramétrique à la ML.

Polymorphisme ad hoc

Ce système est différent du polymorphisme vu jusqu'à présent. En ML une fonction polymorphe ne regarde pas ses arguments

polymorphes. Le traitement est identique pour tous les types. En Haskell c'est le contraire. Une fonction polymorphe peut avoir un comportement différent selon le type de ses arguments polymorphes. Cela autorise la surcharge de fonctions.

L'idée de base est de définir des types de classes qui regroupent des ensembles de fonctions surchargées. Une déclaration de classe définit une nouvelle classe et les opérateurs que celle-ci autorise. Une déclaration d'instance (d'une classe) indique qu'un certain type est une instance d'une classe. Cela inclut la définition des opérateurs surchargés de sa classe pour ce type.

Par exemple la classe `Num` a la déclaration suivante :

```
class Num a where
    (+)      :: a -> a -> a
    negate  :: a -> a
```

On peut maintenant déclarer une instance `Int` de la classe `Num` de cette manière :

```
instance Num Int where
    x + y      = addInt x y
    negate x = negateInt x
```

Et l'instance `Float` :

```
instance Num Float where
    x + y      = addFloat x y
    negate x = negateFloat x
```

L'application de `negateNum` aura un comportement différent si l'argument est de l'instance `Int` ou `Float`.

L'autre intérêt des classes provient de l'héritage entre classes. La classe descendante récupère les fonctions déclarées par son ancêtre. Ses instances peuvent en modifier le comportement.

Autres caractéristiques

Les autres caractéristiques du langage Haskell sont principalement les suivantes :

- un système d'entrées-sorties purement fonctionnel utilisant des monades;
- les tableaux sont construits paresseusement;
- les vues qui permettent différentes représentations d'un même type de données.

En fait il contient à peu près tous les traits pointus issus de la recherche dans le domaine des langages fonctionnels. C'est son avantage et son inconvénient.

Langages de communication

ERLANG

ERLANG est un langage fonctionnel typé dynamiquement pour la programmation concurrente. Il a été développé par la société Ericsson dans le cadre d'applications pour les télécommunications. Il est maintenant en open source. Le site principal d'accès au langage est le suivant :

Lien: <http://www.erlang.org>

Il est conçu pour que la création de processus et leur communication soient aisées. Les communications se font par envoi de messages et elles peuvent être soumises à des délais. Il est facile de définir des protocoles via des ports. Chaque processus possède son propre dictionnaire de définition. La gestion des erreurs utilise un mécanisme d'exceptions et de signaux qui peuvent se propager entre les processus. De nombreuses applications de téléphonie ont été réalisées avec Erlang, faisant gagner un temps de développement non négligeable.

SCOL

Le langage SCOL est un langage de communication pour la

construction de mondes 3D. Il est développé par la société Cryo Networks :

Lien: <http://www.cryo-networks.com>

Son noyau est proche de celui de Caml : il est fonctionnel, statiquement typé, polymorphe paramétrique avec inférence de types. Il est « multimedia » grâce à ses API pour le son, la 2D et la 3D. Le moteur 3D est très efficace. L'originalité de SCOL provient de la communication entre machines virtuelles au travers de canaux. Un canal est un couple (environnement, liaison réseau). La liaison est une socket (TCP ou UDP).

L'originalité de SCOL est d'avoir résolu simplement le problème de la sécurisation de code téléchargé : seul le texte des programmes circule sur le réseau. La machine réceptrice type le programme passé puis l'exécute garantissant que le code produit provient bien du compilateur officiel. Pour mettre en oeuvre une telle solution, sans sacrifier la vitesse de transfert et de réception, le choix d'un langage fonctionnel statiquement typé s'est imposé pour la concision des sources qu'il autorise.

Langages à Objet : comparaison avec Java

Bien qu'Objective CAML soit issu du monde fonctionnel, il est nécessaire de comparer son extension objet à un représentant important des langages à objets. Nous choisissons le langage Java qui bien que proche du point de vue de son implantation, diffère fortement par son modèle objet et par son système de types.

Le langage Java est un langage à objets développé par la société SUN. Le site principal d'accès au langage est le suivant:

Lien: <http://java.sun.com>

Principales caractéristiques

Le langage Java est un langage à classes. L'héritage est simple et permet de redéfinir ou de surcharger des méthodes héritées. Le

typage est statique. Une classe héritée est en relation de sous-typage avec sa classe ancêtre.

Java ne possède pas de classe paramétrée. On obtient deux types de polymorphisme : ad hoc pour la surcharge et d'inclusion pour la redéfinition.

Il est multi-threading et permet de développer des applications réparties soit en utilisant les sockets soit en invoquant une méthode sur des objets distants (Remote Method Invocation).

Les principes de son implantation sont proches de celle d'Objective CAML. Un programme Java est compilé vers une machine abstraite (JVM). Le chargement du code est dynamique. Le code produit est indépendant des architectures machines en étant évalué par un interprète de la machine virtuelle. Les types de données de base sont spécifiés de sorte à garantir une même représentation sur toutes les architectures. La bibliothèque d'exécution est munie d'un GC.

Java possède d'importantes bibliothèques de classes (environ 600 avec le JDK auxquelles s'ajoutent les nombreux développements indépendants). Les principales bibliothèques concernent les interfaces graphiques et les opérations d'entrées-sorties intégrant la communication entre machines.

Différences avec Objective CAML

Les principales différences entre Java et Objective CAML tiennent à leur système de type, à la redéfinition et à la surcharge des méthodes. La redéfinition d'une méthode héritée doit utiliser des paramètres exactement de même type. La surcharge d'une méthode permet d'aiguiller le choix de la méthode à employer selon les types des paramètres d'appel. Dans l'exemple suivant la classe B hérite de la classe A. La classe B redéfinit la première version de la méthode `to_string`, mais surcharge la deuxième version. De même la méthode `eq` est surchargée car le type du paramètre (ici B) n'est pas égal au type du paramètre de la méthode héritée (ici A). Au final la classe B possède deux méthodes `eq` et deux méthodes `to_string`.

```

class A {
    boolean eq (A o) { return true;}
    String to_string (int n ) { }
}

class B extends A {
    boolean eq (B o) { return true;}
    String to_string (int n ) { }
    String to_string (float x, float y)
}

```

Bien que la liaison soit retardée, la résolution de la surcharge, c'est-à-dire la détermination du type de la méthode à utiliser, est effectuée à la compilation.

La deuxième différence importante provient de la possibilité de forcer le type d'un objet comme on le ferait en C. Dans l'exemple suivant, on définit deux objets a et b, respectivement de classe A et B. On déclare ensuite trois variables c, d et e en posant une contrainte de type sur les valeurs affectées.

```

{
    A a = new A ();
    B b = new B ();
    A c = (A) b;
    B d = (B) c;
    B e = (B) a;
}

```

Comme le type de b est sous-type du type de a, alors l'affectation de b à c est acceptée. Dans ce cas la contrainte de type peut être omise. Par contre les deux contraintes suivantes nécessitent d'effectuer un test dynamique de type pour garantir que les valeurs dans c et dans a correspondent en fait à des objets de la classe B. Dans ce programme cela est vrai pour c, mais faux pour a. Ce dernier cas déclenche alors une exception. Bien que cela soit pratique, en particulier pour les interfaces graphiques, ces contraintes de type peuvent entraîner des déclenchements d'exceptions à l'exécution pour l'utilisation de types erronés. En cela Java est un langage typé en partie statiquement et en partie dynamiquement. De plus l'absence de classe paramétrée oblige bien souvent à utiliser ce trait

pour écrire des classes génériques.

Avenir des développements en Objective CAML

Il est difficile pour un nouveau langage d'exister s'il ne s'accompagne pas d'un développement important d'une application (comme Unix pour C) ou d'un support commercial et industriel conséquent (comme SUN pour JAVA). Les qualités propres du langage sont rarement suffisantes. Objective CAML possède de nombreuses qualités et quelques défauts que nous avons décrits au cours de ce chapitre. Pour sa part, Objective CAML est soutenu par l'INRIA où il est conçu et implanté au sein du projet CRISTAL. Issu de la recherche académique, Objective CAML y est utilisé comme laboratoire expérimental pour tester de nouveaux paradigmes de programmation et comme langage d'implantation. Il est largement enseigné dans les différents cycles universitaires et dans les classes préparatoires. Plusieurs milliers d'étudiants et d'élèves apprennent chaque année les concepts du langage et le pratiquent. En cela le langage Objective CAML a une place importante dans le monde académique. L'enseignement de l'informatique, en France, mais aussi aux Etats Unis, forme de nombreux programmeurs à ce langage tant au niveau pratique que théorique.

En revanche, dans le monde industriel le mouvement est moins dynamique. À notre connaissance, il n'y a pas une seule application commerciale, développée en Objective CAML, vendue au grand public et vantant l'utilisation d'Objective CAML. Le seul exemple approchant est celui du langage SCOL de chez Cryo-Networks. Il y a pourtant un léger frémissement en ce sens. Les premiers appels d'offres à des SSII d'applications en Objective CAML apparaissent. Sans espérer rapidement un effet boule de neige, il est significatif qu'une demande existe pour ce type de langage. Et sans espérer non plus un retour sur investissement à très court terme, il est important de s'y intéresser.

C'est maintenant au langage et à son environnement de développement de montrer leur pertinence. Pour accompagner ce phénomène, il est sans doute nécessaire d'apporter certaines garanties quant à l'évolution du langage. À ce titre, Objective CAML n'est qu'encore émergent et doit faire le choix de sortir plus avant

du monde académique. Mais cette « sortie dans le monde » n'aura d'effet que si certaines règles sont respectées :

- garantir la pérennité des développements en assurant la compatibilité ascendante des prochaines versions du langage (la difficulté étant la stabilité des nouveaux éléments (objet, etc));
- spécification du langage en liaison avec les développeurs actuels en vue d'une future normalisation (ce qui permettrait la réalisation de plusieurs implantations pour garantir l'existence de plusieurs solutions) ;
- concevoir un environnement de développement contenant une interface graphique portable, un bus CORBA, des interfaces avec les SGBD, et surtout un environnement de mise au point plus convivial.

Certains des points évoqués en particulier la normalisation peuvent rester du ressort de l'académie. D'autres n'offrent d'intérêt que pour le monde industriel. Tout dépendra alors de leur degré d'entente. Il existe un précédent montrant qu'un langage peut être « libre » et pourtant maintenu commercialement comme ce fut le cas pour le compilateur gnat du langage ADA et la société ACT.

Lien: <http://www.act-europe.fr>

Conclusion

Bien que l'informatique soit devenue une activité industrielle, à bien des égards le succès d'un langage de programmation est affaire de subjectivité. Si < < le coeur a ses raisons que la raison ignore > > alors Objective CAML est un choix raisonnable pour un amant de coeur.

Il repose sur des fondements théoriques solides tout en disposant d'un large spectre de paradigmes de programmation. Si on ajoute la simplicité de la prise de contact avec le langage que permet la boucle d'interaction, cela en fait un langage parfaitement adapté à l'enseignement.

- Les types structurés et les types abstraits permettent d'aborder

les problèmes d'algorithmique et leurs structures de données complexes tout en s'abstrayant des problèmes de représentation mémoire et d'allocation.

- Le modèle théorique fonctionnel sous-jacent au langage fournit une introduction précise aux notions d'évaluation et de typage dont « l'honnête programmeur » se doit d'être instruit.
- Les différents modèles de programmation peuvent être abordés indépendamment les uns des autres : de la structuration modulaire ou par objets des logiciels à la programmation système de bas niveau, il est peu de domaines où Objective CAML ne soit pas pertinent.
- Son adéquation avec la programmation symbolique en fait un excellent support pour des enseignements théoriques comme la compilation ou l'intelligence artificielle.

Pour ces qualités, Objective CAML est souvent utilisé comme support d'enseignement à l'initiation à l'informatique aussi bien que pour des cours de programmation avancée qui explicitent le lien entre le haut niveau d'abstraction du langage et son exécution. Nombre d'enseignants ont été et restent séduits par l'intérêt pédagogique d'Objective CAML et, par voie de conséquence, nombre d'informaticiens y ont été formés.

Un des premiers sujets de satisfaction du développement en Objective CAML est son confort d'utilisation. Le compilateur se charge rapidement et son inférence statique de types ne laisse rien échapper. D'autres analyses statiques du code donnent de précieux indices d'anomalies sinon d'erreurs pour le programmeur : les filtrages incomplets sont signalés, l'application partielle d'une fonction dans une séquence est détectée, etc. À ce premier sujet de satisfaction s'en ajoute un second : le compilateur engendre très rapidement un code efficace.

Performance du compilateur, concision d'expression de la programmation fonctionnelle, qualité et diversité des bibliothèques font d'Objective CAML un langage parfaitement adapté aux besoins du < < logiciel jetable > >. Mais ce serait le réduire que de le cantonner à ce seul domaine d'application. Pour ces mêmes raisons, Objective CAML est un outil précieux d'expérimentation et de

prototypage d'applications. De plus, lorsqu'aux traits déjà cités viennent s'ajouter les mécanismes de structuration des modules et des objets, le langage ouvre la voie à la conception et au développement d'applications abouties.

Enfin, Objective CAML et sa communauté de développeurs forment un milieu extrêmement réactif à l'innovation dans le domaine de la programmation. La libre disponibilité et la diffusion des sources du langage offrent un terrain d'expérimentation ouvert aux concepts émergeants.

L'apprentissage d'Objective CAML réclame un certain effort au programmeur familier d'autres langages. Et ce, alors même que l'objet de son étude est en évolution constante. Nous espérons que sans masquer la complexité de certains concepts, cet ouvrage facilitera cette phase d'apprentissage et pourra ainsi accélérer le retour sur investissement du développeur d'applications en Objective CAML.

Part V Annexes

Annexe A Types cycliques

Introduction

Le système de type d'Objective CAML serait beaucoup plus simple si le langage était purement fonctionnel. Par « malheur » les extensions du langage introduisent des extensions du langage de type et du mécanisme d'inférence : nous en avons eu l'illustration avec les variables de type faibles (voir page ??) rendues nécessaires par les extensions impératives.

Le typage des objets introduit une notion de type cyclique associée au mot clé **as** (voir page ??) qui peut être utilisée hors du concept de la programmation par objet. Cette annexe décrit cette extension du langage de type qui est accessible par une option du compilateur.

Types cycliques

Objective CAML permet de déclarer des structures de données récursives, c'est à dire des structures pouvant comporter une valeur ayant la même structure.

```
# type somme_ex1 = Ctor of somme_ex1 ;;
type somme_ex1 = | Ctor of somme_ex1
# type record_ex1 = { champ : record_ex1 } ;;
type record_ex1 = { champ: record_ex1 }
```

La construction de valeurs de ces types n'est pas immédiate car il est nécessaire de posséder une valeur pour en construire une. La déclaration récursive de valeurs permet de se tirer de cercle vicieux.

```
# let rec somme_val = Ctor somme_val ;;
val somme_val : somme_ex1 = Ctor (Ctor (Ctor (Ctor (Ctor
# let rec val_record_1 = { champ = val_record_2 }
  and val_record_2 = { champ = val_record_1 } ;;
val val_record_1 : record_ex1 = {champ={champ={champ={ch
val val_record_2 : record_ex1 = {champ={champ={champ={ch
```

Les arbres planaires généraux peuvent se représenter par une structure de données de cette forme.

```
# type 'a arbre = Sommet of 'a * 'a arbre list ;;
type 'a arbre = | Sommet of 'a * 'a arbre list
# let hauteur_1 = Sommet (0,[]) ;;
val hauteur_1 : int arbre = Sommet (0, [])
# let hauteur_2 = Sommet (0,[ Sommet (1,[]); Sommet (2,
val hauteur_2 : int arbre =
  Sommet (0, [Sommet (1, []); Sommet (2, []); Sommet (3,
# let hauteur_3 = Sommet (0,[ hauteur_2; hauteur_1 ]) ;;
val hauteur_3 : int arbre =
  Sommet
  (0,
  [Sommet (0, [Sommet (...); Sommet (...); Sommet (...)]
```

(* idem avec un record *)

```
# type 'a arbre_rec = { etiquette:'a ; fils:'a arbre_rec
type 'a arbre_rec = { etiquette: 'a; fils: 'a arbre_rec
# let haut_rec_1 = { etiquette=0; fils=[] } ;;
val haut_rec_1 : int arbre_rec = {etiquette=0; fils=[]}
# let haut_rec_2 = { etiquette=0; fils=[haut_rec_1] } ;;
val haut_rec_2 : int arbre_rec = {etiquette=0; fils=[{et
```

On pourrait se dire qu'il n'est pas utile d'avoir un type énuméré avec un seul constructeur mais, par défaut, Objective CAML n'accepte pas les abréviations de types récursives.

```
# type 'a arbre = 'a * 'a arbre list ;;
Characters 7-36:
The type abbreviation arbre is cyclic
```

On peut définir des valeurs ayant cette structure, mais elles n'ont pas le même type.


```
# let arbre_1 = (0,[]) ;;
val arbre_1 : int * 'a list = 0, []
# let arbre_2 = (0,[ (1,[]); (2,[]); (3,[]) ]) ;;
val arbre_2 : int * (int * 'a list) list = 0, [1, []; 2, []; 3, []]
# let arbre_3 = (0,[ arbre_2; arbre_1 ]) ;;
val arbre_3 : int * (int * (int * 'a list) list) list =
  0, [0, [...; ...; ...]; 0, []]
```

De la même manière, Objective CAML n'est pas capable d'inférer un type à une fonction prenant en argument une valeur de cette forme.

```
# let max_list = List.fold_left max 0 ;;
val max_list : int list -> int = <fun>
# let rec hauteur = function
  Sommet (_,[]) -> 1
  | Sommet (_,fils) -> 1 + (max_list (List.map hauteur fils))
val hauteur : 'a arbre -> int = <fun>
# let rec hauteur2 = function
  (_,[]) -> 1
  | (_,fils) -> 1 + (max_list (List.map hauteur2 fils)) ;
```

Characters 97-101:

```
This expression has type 'a list but is here used with type
('b * 'a list) list
```

Le message d'erreur nous montre que la fonction hauteur2 serait typable si nous avions l'égalité entre les types *'a* et *'b * 'a list*. C'est justement cette égalité qui nous a été refusée dans la déclaration de l'abréviation de type *arbre*.

Pourtant, le typage des objets permet de construire des valeurs dont le type est cyclique. Examinons la fonction suivante et tentons de deviner son type.

```
# let f x = x#copy = x ;;
```

Le type de *x* est une classe qui possède la méthode *copy*. Le type de cette méthode est le même que celui de *x* puisque l'on teste leur égalité. En conséquence, si *foo* est le type de *x* il est de la forme *<copy : foo ; .. >*. D'après ce que nous avons vu plus haut, son type étant cyclique, cette fonction devrait être rejetée; et pourtant il n'en est rien.

```
# let f x = x#copy = x ;;
val f : (< copy : 'a; .. > as 'a) -> bool = <fun>
```

Objective CAML accepte cette fonction et dénote la cyclicité du type par le **as** qui identifie 'a avec un type contenant 'a.

En réalité, les deux problèmes sont exactement les mêmes, mais par défaut, Objective CAML n'accepte ce genre de types que lorsqu'ils concernent des objets. La fonction hauteur est typable si elle produit une cyclicité sur le type d'un objet.

```
# let rec hauteur a = match a#fils with
  [] -> 1
  | 1 -> 1 + (max_list (List.map hauteur l)) ;;
val hauteur : (< fils : 'a list; .. > as 'a) -> int = <f
```

Option

Il existe une option du compilateur et de la boucle d'interaction qui permet de lever cette limitation aux objets des types cycliques.

```
$ ocamlc -rectypes ...
$ ocamlpt -rectypes ...
$ ocaml -rectypes
```

Si nous reprenons les exemples précédents dans une boucle d'interaction lancée avec cette option, voici ce qu'on obtient.

```
# type 'a arbre = 'a * 'a arbre list ;;
type 'a arbre = 'a * 'a arbre list
# let rec hauteur = function
  (_, []) -> 1
  | (_, fils) -> 1 + (max_list (List.map hauteur fils)) ;;
val hauteur : ('b * 'a list as 'a) -> int = <fun>
```

Les valeurs arbre_1, arbre_2 et arbre_3 définies auparavant n'ont pas le même type mais ont toutes un type compatible avec celui de hauteur.

```
# hauteur arbre_1 ;;
- : int = 1
```

```
# hauteur arbre_2 ;;
- : int = 2
# hauteur arbre_3 ;;
- : int = 3
```

Le mot clé **as** fait partie du langage de type, à ce titre il peut être utilisé dans une déclaration de type.

Syntaxe

typenom = *typedef* **as** *var*;; Nous pouvons définir le type *arbre* en utilisant cette syntaxe.

```
# type 'a arbre = ( 'a * 'sommet list ) as 'sommet ;;
type 'a arbre = 'a * 'a arbre list
```

Warning

Si ce mode peut se révéler utile en quelques occasions, il a tendance à accepter le typage de trop de valeurs en leur donnant des types difficiles à lire.

Sans l'option `-rectypes`, la fonction suivante aurait été rejetée par le typeur.

```
# let inclus l1 l2 =
  let rec mem x = function
    [] -> false
    | a::l -> (l=x) || (mem x a) (* erreur volontaire : inv
  in List.for_all (fun x -> mem x l2) l1 ;;
val inclus : ('a list as 'a) list list -> ('b list as 'b
```

Bien que l'examen du type permet de constater immédiatement qu'il y a une erreur, il n'y a plus de message d'erreur permettant de localiser cette erreur.

Annexe B Objective CAML 2.99

Introduction

Indépendamment du développement d'Objective CAML, de nombreuses extensions du langage sont apparues. L'une d'entre-elles, baptisée O'Labl, est en cours d'intégration dans Objective CAML et sera sans doute présente dans les prochaines versions Objective CAML 3.xx. Une version de test est distribuée sous le nom de Objective CAML 2.99; elle est présente sur le cédérom accompagnant cet ouvrage.

Lien: <http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/>

Cette annexe est consacrée à une rapide illustration des nouvelles possibilités apportées par :

- les labels¹;
- les arguments optionnels;
- les constructeurs polymorphes;
- l'interface de développement ocamlbrowser;
- la bibliothèque LablTk.

Cette version étant en cours d'évaluation, elle n'est pas exempte de bogues et elle est susceptible d'évoluer dans la version définitive.

Extensions du langage

O'Labl apporte trois extensions à Objective CAML. Deux d'entre elles ne modifient pas le reste du langage dans le sens où un programme écrit dans la version 2.04 conserve le même comportement dans la version 2.99. La troisième extension et les modifications qui ont été apportées aux bibliothèques standards sont susceptibles de changer le sens des programmes écrits sans tenir compte des nouveaux traits du langage. Pour assurer une compatibilité avec le passé, il existe deux modes distincts pour utiliser Objective CAML 2.99 :

1. Le mode **classique** : c'est l'utilisation par défaut du compilateur et de la boucle d'interaction. L'extension constituée par les labels est restreinte pour ne pas modifier la sémantique des programmes qui ne les utilisent pas.
2. Le mode **moderne** : il est choisi explicitement. L'extension

des labels permet plus de choses, mais l'utilisation des bibliothèques demande d'utiliser les spécificités de ce nouveau trait.

Le compilateur et la boucle d'interaction possèdent une nouvelle option (-modern) pour explicitement utiliser les nouveaux traits des labels. Par défaut, le style **classique** restreint les possibilités des labels et conserve la compatibilité avec les compilateurs précédents.

```
$ ocamlc -modern .....  
$ ocaml -modern .....
```

La boucle d'interaction autorise de commuter les deux modes pendant son utilisation en utilisant la directive **#modern**.

```
# #modern true ;;    (* passe en mode moderne *)  
# #modern false ;;   (* passe en mode classique *)
```

Warning

Un programme correct en 2.04, même s'il n'utilise aucune des extensions de la 2.99, peut ne plus être correct dans la nouvelle version.

Labels

Un est une annotation apportée aux arguments d'une fonction dans sa déclaration et dans son application. Il se présente comme un identificateur séparé du paramètre (formel ou effectif) d'une fonction par le symbole ':'.

On trouve les labels dans les déclarations de fonctions :

Syntaxe

let*fonctionlabel:p = exp* dans les déclarations anonymes avec le mot clé **fun** :

Syntaxe

fun*label:p -> exp* et dans le paramètre effectif d'une fonction :

Syntaxe

(fonctionlabel:exp)

labels dans les types

Les labels donnés aux arguments d'une expression fonctionnelle apparaissent dans son type et annotent les types des arguments auxquels ils réfèrent.

```
# let add op1:x op2:y = x + y ;;  
val add : op1:int -> op2:int -> int = <fun>  
# let mk_triplet arg1:x arg2:y arg3:z = (x,y,z) ;;  
val mk_triplet : arg1:'a -> arg2:'b -> arg3:'c -> 'a * 'b * 'c
```

Si on souhaite donner le même identificateur au label et à la variable, il n'est pas utile de le répéter car c'est celui pris par défaut.

Syntaxe

fun:*p -> exp*

```
# let mk_triplet :arg1 :arg2 :arg3 = (arg1,arg2,arg3) ;;  
val mk_triplet : arg1:'a -> arg2:'b -> arg3:'c -> 'a * 'b * 'c
```

L'utilisation du caractère **:** dans les déclarations de labels impose d'utiliser des espaces pour les autres utilisations de ce symbole à des endroits où il pourrait y avoir ambiguïté.

```
# let id x:int = int ;;    (* x est le label du paramètre *)
```

```
val id : x:'a -> 'a = <fun>  
# let id x : int = x ;;    (* x est un paramètre de type  
val id : int -> int = <fun>
```

Warning

La contrainte d'un type dans une expression doit se faire en respectant la syntaxe des déclarations de labels.

Il n'est pas autorisé de définir des labels dans une déclaration de fonction par filtrage; en conséquence le mot clé **function** ne peut pas être utilisé pour une fonction avec label.

```
# let f = function arg:x -> x ;;  
Characters 18-22:  
Syntax error  
# let f = fun arg:x -> x ;;  
val f : arg:'a -> 'a = <fun>
```

labels en mode classique

Dans un premier temps nous présentons les labels dans le mode **classique**.

Application

Les labels peuvent être utilisés pour appliquer les arguments à une fonction, mais ils ne sont pas obligatoires.

```
# mk_triplet '1' 2 3.0 ;;  
- : char * int * float = '1', 2, 3  
# mk_triplet arg1:'1' arg2:2 arg3:3.0 ;;  
- : char * int * float = '1', 2, 3  
# mk_triplet '1' arg2:2 3.0 ;;  
- : char * int * float = '1', 2, 3
```

Les labels ne modifient pas les possibilités d'application partielle.

```
# let couple = mk_triplet arg1:() ;;  
val couple : arg2:'_a -> arg3:'_b -> unit * '_a * '_b =
```

Warning

Dans le mode **classique**, les arguments d'une fonction doivent être appliqués dans l'ordre de leur déclaration.

```
# let couple = mk_triplet arg2:() ;;  
Characters 30-32:  
Expecting function has type arg1:'a -> arg2:'b -> arg3:'  
This argument cannot be applied with label arg2:
```

Lisibilité du code

L'intérêt principal des labels est d'associer au type d'un argument une annotation qui rend plus explicite l'interface d'une fonction.

```
# String.sub ;;  
- : string -> pos:int -> len:int -> string = <fun>
```

Plus aucun doute, la fonction `String.sub` prend en argument une chaîne, la position du premier caractère, et la longueur de la chaîne à extraire.

Les fonctions des bibliothèques de la distribution d'Objective CAML 2.99 ont été « labelisées ». Le tableau [B.1](#) donne les conventions qui ont été utilisées.

label	signification
pos:	une position (dans une liste, une chaîne, un tableau, etc.)
len:	une taille (length)
buf:	une chaîne utilisée comme tampon (buffer)
src:	la source d'une opération

dst:	la destination d'une opération
fun:	une fonction à appliquer
pred:	un prédicat
acc:	un accumulateur
to:	un canal de sortie (<i>out_channel</i>)
key:	une clé utilisée dans un index (liste d'associations, etc.)
data:	une valeur associée utilisée dans un index
mode:	un mode ou une liste d'options
perm:	des permissions de fichiers

Figure B.1 : conventions sur les labels

Labels en mode moderne

La principale différence avec le mode **classique** est que dans le mode **moderne**, il est obligatoire de donner explicitement les labels lors de l'application d'une expression fonctionnelle.

```
# let add :x :y = x + y ;;
val add : x:int -> y:int -> int = <fun>
# #modern false ;;
# add 1 2 ;;
- : int = 3
# #modern true ;;
# add 1 2 ;;
```

Characters 4-5:

```
Expecting function has type x:int -> y:int -> int
This argument cannot be applied without label
```

Warning

Dans le mode **moderne**, il est obligatoire de donner le nom des labels aux arguments qui en possèdent un.

La conséquence de cette obligation est que l'ordre des arguments possédant un label devient pour le coup indifférent puisqu'on peut les identifier par leur label. Objective CAML 2.99 possède un système permettant de commuter les arguments d'une fonction et donc de les appliquer dans n'importe quel ordre.

```
# add x:1 y:2 ;;
- : int = 3
# add y:2 x:1 ;;
- : int = 3
```

C'est particulièrement utile quand on souhaite faire l'application partielle d'un argument qui n'est pas le premier de la déclaration.

```
# let add_x_2 = add y:2 ;;
val add_x_2 : x:int -> int = <fun>
# add_x_2 x:1 ;;
- : int = 3
```

Les arguments possédant le même label ou n'ayant pas de label ne commutent pas entre eux. Une application dans un tel cas considère le premier argument qui a ce label (respectivement qui n'a pas de label).

```
# let test arg1:_ arg2:_ _ arg2:_ _ = () ;;
val test : arg1:'a -> arg2:'b -> 'c -> arg2:'d -> 'e ->
# test arg2:() ;; (* le premier label arg2: de la déclara
- : arg1:'a -> 'b -> arg2:'c -> 'd -> unit = <fun>
# test () ;;      (* le premier argument sans label de
- : arg1:'a -> arg2:'b -> arg2:'c -> 'd -> unit = <fun>
```

Paramètres optionnels

Objective CAML 2.99 autorise la définition de fonctions avec des arguments labélisés optionnels. De tels arguments sont définis avec une valeur par défaut qui est la valeur donnée au paramètre si l'application n'en donne pas une autre explicitement.

Syntaxe

fun?(*nom*:*p* = *exp1*) -> *exp2* Le caractère optionnel de l'argument apparaît dans son type par le symbole ?.

```
# let sp_incr ?(inc:x=1) y = y := !y + x ;;
val sp_incr : ?inc:int -> int ref -> unit = <fun>
```

La fonction `sp_incr` se comporte comme la fonction `incr` du module `Pervasives`.

```
# let v = ref 4 in sp_incr v ; v ;;
- : int ref = {contents=5}
```

Mais on peut lui spécifier un incrément différent de celui par défaut.

```
# let v = ref 4 in sp_incr inc:3 v ; v ;;
- : int ref = {contents=7}
```

L'application d'une fonction se fait en donnant la valeur par défaut de tous les paramètres optionnels jusqu'au paramètre effectivement passé par l'application. Si l'argument de l'appel est donné sans label, il est considéré comme étant le premier argument de la fonction qui ne soit pas optionnel.

```
# let f ?(:x1=0) ?(:x2=0) x3 x4 = 1000*x1+100*x2+10*x3+x4
val f : ?x1:int -> ?x2:int -> int -> int -> int = <fun>
# f 3 ;;
- : int -> int = <fun>
# f 3 4 ;;
- : int = 34
# f x1:1 3 4 ;;
```

```
- : int = 1034
# f x2:2 3 4 ;;
- : int = 234
```

Les arguments optionnels peuvent être passés dans un ordre arbitraire quel que soit le mode en cours.

```
# #modern false ;;
# f x2:2 x1:1 3 4 ;;
- : int = 1234
```

Un argument optionnel peut être donné sans valeur par défaut, dans ce cas il est considéré dans le corps de la fonction comme étant du type *'a option*; *None* est sa valeur par défaut.

Syntaxe

fun?*nom*:*p* -> exp

```
# let print_entier ?file:x n =
  match x with
  None -> print_int n
  | Some f -> let fic = open_out f in
    output_string fic (string_of_int n) ;
    output_string fic "\n" ;
    close_out fic ;;
val print_entier : ?file:string -> int -> unit = <fun>
```

Par défaut, la fonction `print_entier` affiche son argument sur la sortie standard. Si on lui passe un nom de fichier avec le label `file`, la sortie se fait alors dans ce fichier.

Remarque

Si les derniers paramètres d'une fonction sont optionnels, il devront être appliqués explicitement.

```
# let test ?x ?y n ?a ?b = n ;;
```

```

val test : ?x:'a -> ?y:'b -> 'c -> ?a:'d -> ?b:'e -> 'c
# test 1 ;;
- : ?a: '_a -> ?b: '_b -> int = <fun>
# test 1 b:'x' ;;
- : ?a: '_a -> int = <fun>
# test 1 a:() b:'x' ;;
- : int = 1

```

Labels et objets

Les labels peuvent être utilisés pour les paramètres d'une méthode ou pour ceux du constructeur d'un objet.

```

# class point ?(:x=0) ?(:y=0) (col : Graphics.color) =
object
  val pos = (x,y)
  val couleur = col
  method affiche ?(to:file=stdout) () =
    output_string file "point (" ;
    output_string file (string_of_int (fst pos)) ;
    output_string file "," ;
    output_string file (string_of_int (snd pos)) ;
    output_string file ")\n"
  end ;;
class point :
  ?x:int ->
  ?y:int ->
  Graphics.color ->
  object
    val couleur : Graphics.color
    val pos : int * int
    method affiche : ?to:out_channel -> unit -> unit
  end
# let obj1 = new point x:1 y:2 Graphics.white
  in obj1#affiche () ;;
point (1,2)
- : unit = ()

```

```
# let obj2 = new point Graphics.black
  in obj2#affiche () ;;
point (0,0)
- : unit = ()
```

Les labels et les arguments optionnels fournissent une alternative à la surcharge des méthodes et des constructeurs qui sont classiques dans les langages à objets et qu'Objective CAML ne peut accepter.

Cette émulation de la surcharge comporte quelques limitations, en particulier un des arguments devra ne pas être optionnel; on peut toujours utiliser un argument de type *unit*.

```
# class nombre ?entier ?flottant () =
  object
    val mutable valeur = 0.0
    method print = print_float valeur
    initializer
      match (entier,flottant) with
      | (None,None) | (Some _,Some _) -> failwith "construction"
      | (None,Some f) -> valeur <- f
      | (Some n,None) -> valeur <- float_of_int n
    end ;;
class nombre :
  ?entier:int ->
  ?flottant:float ->
  unit -> object val mutable valeur : float method print
# let n1 = new nombre entier:1 () ;;
val n1 : nombre = <obj>
# let n2 = new nombre flottant:1.0 () ;;
val n2 : nombre = <obj>
```

Constructeurs polymorphes

Les types sommes d'Objective CAML ont deux principales limitations. D'une part il n'est pas possible d'étendre un type somme avec un nouveau constructeur. D'autre part un constructeur ne peut appartenir qu'à un seul type. Objective CAML 2.99 propose des

constructeurs spéciaux dit constructeurs polymorphes qui échappent à ces deux contraintes.

Un constructeur polymorphe est bâti avec un identificateur qui a la même contrainte syntaxique que les autres constructeurs à savoir qu'il doit être préfixé par une majuscule.

Syntaxe

``Nom` ou

Syntaxe

``Nomtype` Un ensemble de constructeurs polymorphes forme un type, mais il n'est pas nécessaire de définir un type pour définir un constructeur polymorphe.

```
# let x = `Entier 3 ;;  
val x : [>`Entier int] = `Entier 3
```

Le type de x avec le symbole [`>`] se lit comme le type qui contient au moins le constructeur ``Entierint`.

```
# let int_of = function  
  `Entier n -> n  
  | `Reel r -> int_of_float r ;;  
val int_of : [<`Entier int|`Reel float] -> int = <fun>
```

A l'inverse, le symbole [`<`] indique que l'argument de `int_of` est du type qui contient au plus les constructeurs ``Entierint` et ``Reelfloat`.

Il est tout de même possible de définir un type par énumération de ses constructeurs polymorphes :

Syntaxe

`typet = [`Nom1|`Nom2] ... [`Nomn]` ou pour des types paramétrés :

Syntaxe

```
type('a,b,...)t= [`Nom1 | `Nom2 | ... | `Nomn]
```

```
# type valeur = [ `Entier int | `Reel float ] ;;  
type valeur = [`Entier int | `Reel float]
```

Les constructeurs polymorphes, comme leur nom l'indique, peuvent prendre des arguments de plusieurs valeurs.

```
# let v1 = `Nombre 2  
  and v2 = `Nombre 2.0 ;;  
val v1 : [>`Nombre int] = `Nombre 2  
val v2 : [>`Nombre float] = `Nombre 2
```

Mais pour autant, v1 et v2 ne sont pas du même type.

```
# v1=v2 ;;  
Characters 4-6:  
This expression has type [>`Nombre float] but is here used  
  [>`Nombre int]
```

D'une manière plus générale, les contraintes sur le type des arguments des constructeurs polymorphes sont accumulées dans leur type par l'annotation &.

```
# let test_nul_entier = function `Nombre n -> n=0  
  and test_nul_reel = function `Nombre r -> r=0.0 ;;  
val test_nul_entier : [<`Nombre int] -> bool = <fun>  
val test_nul_reel : [<`Nombre float] -> bool = <fun>  
# let test_nul x = (test_nul_entier x) || (test_nul_reel x)  
val test_nul : [<`Nombre int & float] -> bool = <fun>
```

Le type de test_nul indique que les seules valeurs acceptées par cette fonction sont celles avec le constructeur *`Nombre* et un argument qui est à la fois de type *int* et de *float*. C'est à dire que les seules valeurs acceptables sont de type 'a !

```
# let f () = test_nul (failwith "rend une valeur de type
```



```
val f : unit -> bool = <fun>
```

Les types des constructeurs polymorphes sont eux-mêmes susceptibles d'être polymorphes.

```
# let id = function `Ctor -> `Ctor ;;  
val id : [<`Ctor] -> [>`Ctor] = <fun>
```

Le type de la valeur de retour de `id` est « les ensembles de constructeurs qui contiennent au moins ``Ctor` » donc c'est un type polymorphe qui peut s'instancier en un type plus précis. De même, l'argument de `id` est « les ensembles de constructeurs qui contiennent au plus ``Ctor` » qui lui aussi est susceptible d'être précisé. En conséquence, ils suivent le mécanisme général des types polymorphes d'Objective CAML à savoir qu'ils sont susceptibles d'être affaiblis.

```
# let v = id `Ctor ;;  
val v : _[>`Ctor] = `Ctor
```

`v` étant le résultat d'une application, son type n'est pas polymorphe (d'où la présence du caractère `_`).

```
# id v ;;  
- : _[>`Ctor] = `Ctor
```

`v` est monomorphe et son type est un sous-type de « contient au moins le constructeur ``Ctor` ». Le fait de l'appliquer à `id` va forcer son type à être un sous-type de « contient au plus le constructeur ``Ctor` ». Logiquement, il doit maintenant avoir le type « contient exactement ``Ctor` ». Ce que nous vérifions.

```
# v ;;  
- : [`Ctor] = `Ctor
```

À la manière des classes, les types des constructeurs polymorphes peuvent être ouverts.

```
# let is_entier = function
```

```

`Entier (n : int) -> true
| _ -> false ;;
val is_entier : [<`Entier int | ..] -> bool = <fun>
# is_entier (`Entier 3) ;;
- : bool = true
# is_entier `Autre ;;
- : bool = false

```

Tous les constructeurs sont acceptés, mais le constructeur ``Entier` doit avoir un argument entier.

```

# is_entier (`Entier 3.0) ;;
Characters 12-23:
This expression has type [>`Entier float] but is here us
[<`Entier int | ..]

```

Toujours comme les classes, les types des constructeurs peuvent être cycliques.

```

# let rec long = function `Rec x -> 1 + (long x) ;;
val long : ([<`Rec 'a] as 'a) -> int = <fun>

```

Pour finir, notons que le type peut être en même temps un sous-ensemble et un sur ensemble de constructeurs. Commençons par un exemple simple.

```

# let ex1 = function `C1 -> `C2 ;;
val ex1 : [<`C1] -> [>`C2] = <fun>

```

Maintenant nous identifions les types d'entrée et de sortie de l'exemple par un second filtre.

```

# let ex2 = function `C1 -> `C2 | x -> x ;;
val ex2 : ([<`C2 | `C1 | .. >`C2] as 'a) -> 'a = <fun>

```

Nous obtenons donc le type ouvert qui contient au moins ``C2` puisque le type de retour contient au moins ``C2`.

```

# ex2 ( `C1 : [> `C1 ] ) ;; (* est bien un sous type de

```

```
- : _[>`C2|`C1] = `C2
# ex2 ( `C1 : [ `C1 ] ) ;;      (* n'est pas un sous type c
Characters 6-9:
This expression has type [`C1] but is here used with typ
```

Bibliothèque

L'interface avec Tcl/Tk a été intégrée à la distribution d'Objective CAML 2.99 mais elle n'a pas été portée sur toutes les plate-formes. L'installation fournit trois nouvelles commandes :

- labltk lance une boucle d'interaction qui intègre la bibliothèque LablTk;
- labltklink s'utilise en lieu et place de ocamlc pour réaliser l'édition de liens de programmes utilisant cette bibliothèque;
- labltkopt est le pendant de labltklink pour la génération de code natif.

Cette bibliothèque définit un grand nombre de modules et utilise massivement les extensions de langage de O'Labl. La présentation de ce module sort du cadre de cette annexe et nous invitons le lecteur intéressé à se référer à la documentation d'Objective CAML 2.99.

Il existe aussi une interface avec Gtk présentée sous la forme d'une hiérarchie de classes, mais elle n'est pas encore dans la distribution. Elle devrait être compatible avec Unix et Windows.

Navigateur OCamlBrowser

OCamlBrowser est un environnement de développement pour Objective CAML développé en utilisant la bibliothèque LablTk ce qui lui permet d'avoir une interface graphique. Il intègre un « navigateur » permettant de se balader entre différents modules, de visualiser leur contenu (noms des valeurs et types) et de les éditer.

Lorsqu'on lance OCamlBrowser par la commande `ocamlbrowser`, on obtient la liste de tous les modules compilés disponibles (voir figure [B.2](#)). On peut en ajouter en donnant les chemins permettant de les trouver. Depuis le menu File, on peut lancer une boucle d'interaction ou un éditeur dans une nouvelle fenêtre.



Figure B.2 : OCamlBrowser : la fenêtre principale

En cliquant sur un des modules, une nouvelle fenêtre s'ouvre affichant son contenu (voir figure B.3). En sélectionnant une valeur, on voit son type apparaître en bas de la fenêtre.

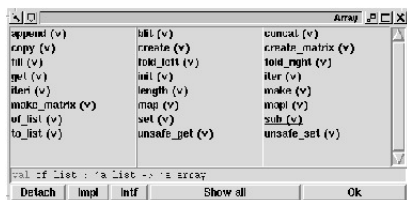


Figure B.3 : OCamlBrowser : contenu d'un module

Depuis la fenêtre principale, on peut faire une recherche sur le nom d'une fonction. Le résultat apparaît dans une nouvelle fenêtre. La figure B.4 montre le résultat de la recherche sur le mot create.



Figure B.4 : OCamlBrowser : recherche sur create

Il y a d'autres possibilités que nous laissons l'utilisateur découvrir.

Références

Bibliographie

[AC96]

Aponte (María-Virginia) et Castagna (Giuseppe). --
 Programmation modulaire avec surcharge et liaison tardive.
In : Journées Francophones des Langages Applicatifs. INRIA. --
 janvier 1996.

[AHU85]

Aho (Alfred), Hopcroft (John) et Ullman (Jeffrey). --
Structures de données et algorithmes. -- InterEditions, 1985.

[And91]

Andrews (G.). -- *Concurrent Programming : Principles and practices*. -- Benjamin Cumming, 1991.

[Ari86]

Ari (Ben). -- *Processus concurrents*. -- Masson, 1986.

[AS89]

Abelson (Harold) et Sussman (Gerald J.). -- *Structures et interprétation des programmes informatiques*. -- InterEditions, 1989.

[ASU89]

Aho (Alfred), Sethi (Ravi) et Ullman (Jeffrey). --
Compilateurs : principes, techniques et outils. -- InterEditions,

1989.

[CCM87]

Cousineau (Guy), Curien (Pierre-Louis) et Mauny (Michel). -- The Categorical Abstract Machine. *Science of Computer Programming*, vol. 8, 1987, pp. 173--202.

[CCS96]

Chailloux (Emmanuel), Cousineau (Guy) et Suárez (Ascánder). -- Programmation fonctionnelle de graphismes pour la production d'illustrations techniques. *Technique et Science Informatiques*, 1996.

[CDM96]

Card (Rémy), Dumas (Éric) et Mével (Franck). -- *Programmation Linux 2.0, API système et fonctionnement du noyau*. -- Eyrolles, 1996, seconde édition.

[Ciz98]

Cizault (Gisèle). -- *IPv6 : Théorie et pratique*. -- O'Reilly, 1998.

[CKL96]

Chailloux (Emmanuel), Kirsch (Laurent) et Lucas (Stéphane). -- Caml2sml, un outil d'aide à la traduction de Caml vers Sml. In : *Journées Francophones des Langages Applicatifs*. INRIA. -- janvier 1996.

[CL99]

Conchon (Sylvain) et Le Fessant (Fabrice). -- JoCaml: mobile agents for Objective-Caml. In : *International Symposium on Agent Systems and Applications*. -- 1999.

[CM95]

Cousineau (Guy) et Mauny (Michel). -- *Approche Fonctionnelle de la Programmation*. -- EdiScience, 1995.

[CP95]

Caspi (Paul) et Pouzet (Marc). -- A functional extension to lustre. In : *8th International Symposium on Languages for*

Intensional Programming. World Scientific. -- Sydney, mai 1995.

[DDL98]

Danelutto (Marco), Di Cosmo (Roberto), Leroy (Xavier) et Pelagatti (Susanna). -- Parallel functional programming with skeletons: the ocamlp3l experiment. In : *ML Workshop*. ACM SIGPLAN. -- 1998.

[DEMN98]

Ducournau (Roland), Euzenat (Jérôme), Masini (Gérald) et Napoli (Amedeo) (édité par). -- *Langages et modèles à objets: état et perspectives de la recherche*. -- INRIA, 1998.

[Eng98]

Engel (Emmanuel). -- *Extensions sûres et praticables du système de types de ML en présence d'un langage de modules et de traits impératifs*. -- Thèse de doctorat, Université Paris-Sud, Orsay, France, mai 1998.

[FC95]

Foisy (Christian) et Chailloux (Emmanuel). -- Caml Flight: a Portable SPMD Extension of ML for Distributed Memory Multiprocessors. In : *Conference on High Performance Functional Computing*. -- avril 1995.

[FF98]

Findler (Robert B.) et Flatt (Matthew). -- Modular Object-Oriented Programming with Units and Mixins. In : *International Conference on Functional Programming*. ACM. -- 1998.

[FGS90]

Froidevaux (Christine), Gaudel (Marie-Claude) et Soria (Michèle). -- *Types de données et algorithmes*. -- Paris, EdiSciences, 1990.

[FW00]

Furuse (Jun) et Weis (Pierre). -- Entrées/Sorties de valeurs en Caml. In : *JFLA'2000 : Journées Francophones des Langages*

Applicatifs. INRIA. -- Mont Saint-Michel, janvier 2000.

[GHJV98]

Gamma (Erich), Helm (Richard), Johnson (Ralph) et Vlissides (John). -- *Design Pattern : catalogue de modèles de conception réutilisables*. -- MIT, 1998.

[HFa96]

Hartel (Pieter), Feeley (Marc) et al. -- Benchmarking implementations of functional languages with ``Pseudoknot'', a float-intensive benchmark. *Journal of Functional Programming*, vol. 6, n 4, 1996.

[HS90]

Harbison (Samuel P.) et Steele (Guy L.). -- *Langage C : manuel de référence*. -- Paris, Masson, 1990, seconde édition.

[Jon98]

Jones (Richard). -- *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. -- John Wiley & Sons, 1998.

[Ler90]

Leroy (Xavier). -- *The ZINC experiment: an economical implementation of the ML language*. -- Technical report n 117, INRIA, 1990.

[Ler92]

Leroy (Xavier). -- *Programmation du système unix en caml light*. -- Rapport technique n RR-147, INRIA, décembre 1992.

[LMB92]

Levine (John R.), Mason (Tony) et Brown (Doug). -- *lex & yacc*. -- O'Reilly, 1992, seconde édition.

[Lou98]

Loulergue (Frédéric). -- BSMML : programmation BSP purement fonctionnelle. In : *RENPAR'10, Rencontres francophones du parallélisme*. Université de Strasbourg. --

Juin 1998.

[LRVD99]

Leroy (Xavier), Rémy (Didier), Vouillon (Jérôme) et Doligez (Damien). -- *The Objective Caml system release 2.04*. -- Rapport technique, INRIA, décembre 1999.

[MdR92]

Mauny (Michel) et de Rauglaudre (Daniel). -- *Parser in ML*. -- Rapport technique n RR-1659, INRIA, avril 1992.

[MNC+89]

Masini (Gérald), Napoli (Amedeo), Colnet (Dominique), Léonard (Daniel) et Tombre (Karl). -- *Les langages à objets*. -- Paris, InterEditions, 1989.

[MT91]

Milner (Robin) et Tofte (Mads). -- *Commentary on Standard ML*. -- MIT, 1991.

[MTH90]

Milner (Robin), Tofte (Mads) et Harper (Robert). -- *The Definition of Standard ML*. -- MIT, 1990.

[Rep92]

Reppy (John). -- *Higher-Order Concurrency*. -- Thèse de PhD, Cornell University, juin 1992.

[Rif90]

Rifflet (Jean-Marie). -- *La communication sous Unix*. -- EdiSciences, 1990.

[Rob89]

Robert (Eric S.). -- *Implementing exceptions in C*. -- Rapport technique n SRC-40, Digital Equipment, 1989.

[Rou96]

Rouaix (François). -- A Web navigator with applets in Caml. *In : Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications*

Networking. pp. 1365--1371. -- Elsevier, May 1996.

[RV98]

Rémy (Didier) et Vouillon (Jérôme). -- Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, vol. 4, n 1, 1998, pp. 27--50. -- A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[Spi90]

Spir (Eric). -- *Gestion Dynamique de la Mémoire dans les Langages de Programmation*. -- InterEdition, 1990.

[Tho99]

Thompson (Simon). -- *Haskell: The Craft of Functional Programming*. -- Addison Wesley, 1999, seconde édition.

[Tur85]

Turner (David A.). -- Miranda: A non-strict functional language with polymorphic types. In : *Proceedings International Conference on Functional Programming Languages and Computer Architecture*, éd. par Jouannaud (J.). pp. 1--16. -- New York, NY, septembre 1985.

[Wil92]

Wilson (Paul. R.). -- Uniprocessors Garbage Collection Techniques. In : *International Workshop on Memory Management*, éd. par Springer-Verlag. ACM SIGPLAN, pp. 1--42. -- septembre 1992.

[Wri93]

Wright (Andrew K.). -- *Polymorphism for Imperative Languages without Imperative Types*. -- Rapport technique n 93-200, Rice University, février 1993.

Index

Index des concepts

- ~~base~~ abstraction, 2
- ~~différenciateur~~, 11
- ~~aggrégation~~, 15
- ~~allocation~~ d'Objective CAML, 1
 - MacOS, 9
 - ~~statique~~ ligne HTML, 1
 - Linux, 1
- analyse lexicale, 11
 - Unix, 1
 - Windows, 1
 - flot, 11
- instance (de classe), 15
- interface, 14, 15
- analyse syntaxique, 11
 - avec C, 12, 12
 - ascendante, 11
 - avec le système, 8
 - conflit, 11, 11
 - graphique, 13
-  shift-reduce, 11
- interopérabilité, 12
- descendante, 11
- label, 8
- flot, 11, 11
- lecteur-écrivain, 19
- ocaml yacc, 11
- liaison
- appel de fonction, voir application
- application, 2
 - statique, 2
- d'une fonction, 2
- ligne de commande, 8
 - partielle, 2
- analyse de, 8
- arbre, 2
- arité, 2
- linéarisation, 8
- liste, 2
- BNF, voir grammaire
- bibliothèque
 - d'association, 8
 - filtrage, 2
 - standard, 8
- bibliothèque, 8
- *Mark&Sweep*, 9
 - préchargée, 8
- mémoire

- bit d'alignement, 9
- booléen, 9
- boucle d'initialisation, 7
 - récupération, 9, 9
 - construction, 7
 - récupération explicite de, 9
 - récupération, 7
 - options, 7
 - explicite, 9
- byte-code, voir code octet
- C (le langage), 92
- callback, 12
- canal de communication, 3, 18
- caractère, 2
 - gestion, 9, 9
- chaîne de caractères, 2, 3
 - récupération, 9
 - représentation, 12
 - automatique, 9
- chargement dynamique, 8
- classe, 15
 - abstraite, 15
 - virtuelle, 15
 - déclaration locale, 15
 - héritage, 15, 7
- machine abstraite, 7
 - héritage multiple, 15
- mise au point, 10, 10
 - instance, 13, 15
- moderne (mode), 3, 3
 - interface, 15
- modifiable, 3
 - paramétrée, 15
- module, 14
 - type, 15
 - type ouvert, 15
 - contrainte, 14
 - variable de, 15
 - déclaration locale, 14
 - virtuelle, 15
 - dépendance, 10
 - ouverture, 8
- classique (mode), 3
 - paramètre, 14
- client-serveur, 20
 - sous-module, 14
- code octet, 7
- motif
 - chargement dynamique, 8
 - compilateur de, voir compilateur
 - combinaison, 2
 - filtrage, 2
- compilateur, 7
 - garde, 2

- intervalle, 10 de caractères, 2
- node, 7, 7
- universelle, 2, 72
- édition de liens, 7
- natif, 7
- profiling, 10
- n-uplet, 2
- notation pointée, 8
- OCaml Browser, 3, 7
- objet, 13, 7
- concurrence, 15, 19
- conditionnelle, 25
- conflit, voir analyse syntaxique
- opérateur, 2, 2
 - associativité, 11
 - fonctionnel, 2
 - priorité, 11, B
- optionnel (argument), B
- couple, 2
- paresseuses (données), 4
- décalage, 6
- pattern matching, voir filtrage
- persistance, 9, 2
- pile d'exécution, 12, 9
- pile d'exécution, 10
- pipe, voir fonction de communication
- pointeur, 3, 2
 - globale simultanée, 2
 - faible, 2
 - locale simultanée, 2
- polymorphisme, 2
 - d'un opérateur, 2
 - récursive, 2
 - d'inclusion, 15
 - de type, 2
- portée d'une variable, 2
- portée, 2, 2
- portabilité, 7
- prise de communication, 20
- de valeur, 2

- débiter, voir mise au point
- destructeur, 9
- digest, voir empreinte
 - léger, 19
- édition de liens, 7
- producteur-consommateur, 19
- produit cartésien, 2
- produit avec C, 12, 2
- profiling, 10
- empreinte, 8
- enregistrement, 2, 10, 10
 - natif, 10, 10
 - champ modifiable, 3
- protocole, 20
- entier, 2
- entrées-sorties, 8, 8
- entrées-sorties
 - avec C, 12
- racine, 9
- environnement, 2, 12
- envoi de message, 15
- ambiguë, 8
- évaluation
- référence, 3
- règle de production, voir grammaire
 - ordre d'évaluation, 3
 - retardée, 4
- Stop&Copy, 9
- événement, 59
- séquentiel autonome, 7
- exception, 19
- sémantique, 19
- self, 15
- session, 7
- affichage, 8
- signal, avec C, 12
- signature, 14
- déclaration, 2
- socket, voir prise de communication
- déclenchement, 2
- sous-typage, 2
- récupération, 13
- stream, voir flot
- exclusion mutuelle, 19
- strict (langage), 4
- expression
- string, voir chaîne de caractères
- structure, 14
 - fonctionnelle, 2
- super, 15
- régulières, 11
- synchronisation, 19
 - rationnelles, 11

- tableau, 3
- famille, 19
- fermeture, 2, 2
- thread, voir processus léger
- top-level, voir boucle d'interaction
- trace, représentation, 12
- tube de communication, 18
- fichier
 - nommé, 18
 - extension
- type
 - .ml, 7
 - abstrait, 147
 - constructeur, 112
 - contrainte, 2, 114, 15, 15
 - cyclique, A
 - interface, 7
 - déclaration, 2
 - enregistrement, 2
- filtrage, 2
 - énuméré, 2
 - fonctionnel, 2, 2
 - exhaustif, 2
 - mutuellement récursif, 2
 - destructif, 4
 - objet, 15
- finalisation, 15, 15
- flot, 4
 - paramétré, 2, 2, 2
 - produit, 2
 - analyse lexicale, 11
 - analyse syntaxique, 11, 11
- flottant, 12
 - représentation, 12
 - mise en notation, 12
 - tableau de, 12
- foncteur, 14
- fonction, 15, 2
- unité de compilation, 7
- unité d'ordre supérieur, 2
 - déclaration avec C, 12
 - définition, 11, 12
 - mutuellement récursives, 2

- partielle, 2
- polymorphe, 2
- valeur
 - trace, 10
 - construction de, 9
 - déclarative, 2
 - déclaration, 2
 - déclaration globale, 2
 - déclaration locale, 2
 - exploration, 10, 12
 - fonctionnelle, 12
 - immédiate, 3, 12
 - partage, 3
 - persistante, 8
- GC, 9, 9, Mark & Sweep 9
 - à génération, 8
 - à racines ambiguës, 9
 - représentation, 12
 - avec C, 12
 - incrémentiel, 9
 - en C, 12
 - majeur, 9
 - mineur, 3, 12
 - structurée, 3, 12
 - Stop&Copy, 9
- variable
- garbage collector, voir GC
- glaneur d'histoires, voir GC
- grammaire, 12
 - liée, 2
 - contextuelle, 11
 - définition, 11
 - de type faible, 3
 - règle de, 11
- vecteur, voir tableau
- grands nombres, 8
- Zinc, 7
- Has-a, 15
- hachage, 7
- heap, 9
- ramasse-misère, 15

Index des éléments du langage

- Lexbuf (type du module Lexing), 11

- `moveto`, 5
- `lisB`(type), 2
- `listenB`, 20, 20
- `loadfile`, 8
- `loadfile_private`, 8
- `lock`, 19
- `log`, 22
- `log10`, 2
- `seek`, 18
- `+`, 2
- `Map` (module), 14
- `Marshal` (module), 8, 8
- `Match_Failure`, 2
- `Mutex` (module), 19
- `major`, 9
- `make`, 8
- `make_image`, 5
- `makelexer`, 11
- `make_matrix`, 8
- `map`, 2, 2, 8, 8
- `map2`, 8, 3
- `mapi`, 2
- `match` (mot-clé), 2, 4
- `matched_string`, 11
- `max_array_length`, 8, 8
- `mem`, 2, 8
- `mem_assoc`, 8
- `mem15sq`, 8
- `memory.h`, 12
- `memq`, 2, 8
- `@method` (mot-clé), 15
- `minor`, 9
- `#mkfile`, 158
- `%values.h`, 12
- `mod`, 2
- `#modern` (directive), B
- `module` (mot-clé), 14
- `module type` (mot-clé), 14
- `mouse_pos`, 5
- `moveto`, 5

- **Argv**(module), 3
- **Arith_status** (module), 8
- **Array** (module), 3, 8, 8, 8
- **Assoc** (module), 8
- **new** (mot-clé), 15
- **add** (available_units), 8
- **add** (interfaces), 8
- **alarm**, 18
- **alloc** (type du module Num), 8
- **allow_unsafe_modules**, 8
- **as** (type), 2, 2
- **object** (mot-clé), 15
- **argv** (commande), 7, 7
- **ocamltype** (commande), B
- **as** (mot-clé), 7, 7, 7
- **ocamlc.opt** (commande), 7
- **ocamldebug** (commande), 10
- **ocamldep** (commande), 10
- **ocamllex** (commande), 11, 11
- **ocamlmktop** (commande), 5, 7, 7
- **ocamlmod** (commande), 7
- **backlight** (commande), 7
- **bigint** (type du module Num), 8
- **bind** (commande), 11, 11
- **bfi** (mot-clé), 2
- **bfi** (image), 3
- **bfd** (type), 2
- **bfs**, 84
- **open** (mot-clé), 8, 14
- **open** (comment), 20
- **open_flag** (type du module Unix), 18
- **Open** (module), 12
- **Open** (module), 19
- **open**, 3
- **open**, 18
- **open** (type), 2
- **open** (type du module Pervasives), 9
- **open**, 8
- **open**, 3
- **class** (mot-clé), 15

- `clear_available_units`, 8
- `clear_graph`, 5
- `ClosestRes2D(module)`, 8
- `Flontrigra(module)`, 8
- `Flontrif(module)`, 8
- `phase_out`, 3
- `plassepr(metsclé)`, 4, 11
- `palott(type)`, 5
- `pipe_line`, 6, 8
- `plotrand`, 8
- `pointactor`, 5
- `pointpileur`
- `print_newline`, 3
- `print_stat`, `modern`, B
- `print_string`, 3
- `concat`, 8, 8
- `print`, 8, 8
- `connect`, 20, 20
- `private` (mot-clé), 15
- `constraint` (mot-clé), 15
- `process_status` (type du module Unix), 18
- `copy`, 8, 15
- `Queue` (module), 8
- `create`, 3, 8, 9, 19, 19, 19
- `create_image`, 5
- `Random` (module), 8
- `create_process`, 18
- `raise` (mot-clé), 2
- `current_point`, 5
- `ratio` (type du module Num), 8
- `read`, 18
- `Delayed`, 4
- `read_key`, 4
- `Digest` (module), 8, 8
- `read_line`, 3
- `DynLink` (module), 8
- `rec` (mot-clé), 2
- `delay`, 19
- `recv`, 19
- `descr_of_in_channel`, 18
- `rectypes`, A
- `descr_of_out_channel`, 18
- `ref` (type), 3
- `do` (mot-clé), 3
- `regexp`, 11
- `done` (mot-clé), 3
- `register`, 12
- `downto` (mot-clé), 3
- `remove`, 8
- `draw_arc`, 5
- `remove_assoc`, 8
- `draw_circle`, 5
- `remove_assq`, 8
- `draw_ellipse`, 5
- `rename`, 8
- `draw_image`, 5
- `rev`, 8
- `dump_image`, 5
- `rev_append`, 8
- `dup`, 18
- `go` (type), 5

- `dup2`, 18
- `Set` (module), 14
- `SOCK_STREAM`, 20
- `Font` (module), 19
- `size` (module), 8
- `snl` (module), 18, 18
- `Spurious` (exception), 4
- `Str` (module), 11
- `Stream` (type du module Unix), 18
- `String` (module), 18
- `System` (module), 8, 20
- `System`, 3
- `exception` (module), 2
- `socks` command (type du module Unix), 18
- `send`, 199
- `serv` (type du module Unix), 20
- `exp`, 8, 9, 9
- `external` (module), 18
- `set_binary_mode_out`, 18
- `File` (module), 8
- `Font` (module), 8
- `fail`, 52
- `false`, 18
- `file`, text size, 5
- `file`, connection, 20
- `file` (module), 14
- `file`, 18
- `file`, 18
- `file`, 18
- `file`, 18, 19
- `file`, behavior (type du module Unix), 18
- `file`, 18
- `file` (type), 2
- `file` of string, 2
- `sleep`, 18
- `file`, 2, 8, 8
- `file` (type du module Unix), 20
- `file` (module), 3
- `file` (type du module Unix), 20
- `file` domain (type du module Unix), 20

- **socket_type** (type du module Unix), 20
- **spin** (type), 8, 8
- **sprintf**, 8
- **smtp_channel**, 8
- **stat_string**, 8
- **status**, 5
- **stdmajor**, 19
- **stdin** (mot-clé), 2
- **stdmotion** (mot-clé), 2
- **structoftype** (mot-clé), 14
- **string**, 8
- **string** (tuple), 29
- **stringof(float)**, 11
- **stringof(int)**, 20
- **string_of_int**, 2
- **structaget** (mot-clé), 14
- **getc**, 8
- **genex**, 19
- **gethostbyaddr**, 20
- **gethostbyname** (module), 19
- **gethostbyname** (module), 20
- **getsetvbyname**, 20
- **ghex** (mot-clé), 20
- **global** (module), 11
- **tl**, 2, 8
- **hashtol** (module), 8, 8
- **handleerror**, 18
- **hchannel**, 8
- **hostentry** (type du module Unix), 20
- **to_string**, 8, 8
- **ifkey** (type du module Genlex), 11
- **#trace** (directive), 10
- **image**, 5
- **try** (mot-clé), 22
- **tryclock**, 19
- **type** (mot-clé), 18
- **inet_addr** (type du module Unix), 20
- **unix** (module), 20
- **unix** (module), 18
- **init** (type), 15

- `imlock`, 319
- `#ifndef` (directive), 10
- `#ifndef_all` (directive), 10
- `int` (type), 2
- `Value_char`, 2
- `val_of_string`, 214, 15
- `intemtable` (mot-clé), 15
- `itane`, 8, 12, 12
- `int2d` (mot-clé), 15
- `iteri`, 8
- `Weak` (module), 8, 9
- `wait_pressed`, 5
- `wait_next_event`, 5
- `wait_time_read`, 20
- `lazy_fimodulo`, 420
- `waitpid` (module), 11
- `write` (mot-clé), 8, 8, 8
- `while` (mot-clé), B
- `while` (mot-clé), 8
- `while` (mot-clé), 8, 14
- `abortsize` (commande), B
- `lazy` (mot-clé), 4
- `length`, 8, 8
- `let` (mot-clé), 2, 2

Parutions O'Reilly

Contacts

Liste de diffusion par messagerie électronique

Si vous souhaitez recevoir périodiquement les annonces de nouveaux produits O'Reilly en français, il vous suffit de souscrire un abonnement à la liste d'annonces

parutions-oreilly

Merci d'expédier en ce cas un message électronique à contenant (dans le corps du message) :

subscribe parutions-oreilly votre_adresse_email

Exemple :

subscribe parutions-oreilly jean.dupond@ici.fr

Cette liste ne véhicule que des annonces et non des *discussions*, vous ne pourrez par conséquent pas y poster. Son volume ne dépasse pas quatre messages par mois.

En cas de problème technique écrire à

parutions-oreilly-owner@ora.de

Site Web

Notre site Web <http://www.editions-oreilly.fr/> diffuse diverses informations :

- le catalogue des produits proposés par les éditions O'Reilly,
- les *errata* de nos ouvrages,
- des archives abritant les exemples,
- la liste des revendeurs.